Grid Application Definition Language

GADL 0.2

Interner Report

Andreas Hoheisel*

18. September 2002

Inhaltsverzeichnis

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

1 Einleitung

Die *Grid Application Definition Language* (GADL) ist eine XML-basierte Beschreibungssprache, die es erlaubt, auf einer abstrakten Ebene komplette Grid-Anwendungen zu beschreiben, mit dem Ziel, aus dieser Beschreibung heraus eine automatische Abbildung auf die verfügbaren Ressourcen zu erzeugen und den Prozessablauf zu steuern. Die GADL wird unter der Führung des Fraunhofer FIRST mit Beiträgen der beteiligten Projektpartner im Rahmen des Projekts *I-Lab* entwickelt [*Pfreundt et al.*, 2001]. Näheres zum Projekt *I-Lab* ist auf der Internetseite des *Fraunhofer Resource Grids* (http://www.fhrg.fhg.de) zu finden.

Die GADL ist gemäß ihrer Aufgabenbereiche in vier Untersprachen gegliedert:

- **GResourceDL:** Die *Grid Resource Definition Language (GResourceDL)* (siehe Kapitel 3.1) dient zur Beschreibung und Kategorisierung der verschiedenen Ressourcen des Fraunhofer Resource Grids. Unter den Begriff Ressourcen fallen hierbei sowohl Hardwareressourcen, als auch Softwarekomponenten und Daten. Benutzerdaten werden zur Zeit bis auf Autorisierungsdaten nicht von der GADL erfasst. Ab GADL 0.2 wird die *Grid Component Definition Language (GCDL)* — die vom Fraunhofer IAO in erster Linie zum Zwecke des Taskmappings entwickelt und bisher getrennt betrachtet wurde — ein fester Bestandteil der GResourceDL.

- **GInterfaceDL:** Softwarekomponenten können Mechanismen vorweisen, die es erlauben, von entfernter Seite aus auf die Funktionalität und den Datenraum dieser Softwarekomponenten zuzugreifen. Sollen nun mehrere Softwarekomponenten innerhalb eines Grid-Jobs interaktiv miteinander gekoppelt werden, so ist es notwendig, diese Schnittstellen zu beschreiben. Dies erfolgt durch die *Grid Interface Definition Language (GInterfaceDL)* (siehe Kapitel 3.2). Die GResourceDL-Beschreibung einer Softwarekomponente wird gegebenenfalls mit der entsprechenden GInterfaceDL-Beschreibung verknüpft.

- **GDataDL:** Die *Grid Data Definition Language (GDataDL)* dient zur Beschreibung von Daten, die als Ressourcen im Grid verfügbar sind oder von Datenströmen, die zwischen Softwarekomponenten innerhalb des Grids ausgetauscht werden (siehe Kapitel 3.3). Die GResourceDL-Beschreibung von Daten wird gegebenenfalls mit der entsprechenden GDataDL-Beschreibung verknüpft.

- **GJobDL:** Mit Hilfe der *Grid Job Definition Language (GJobDL)* werden komplette Grid-Anwendungen beschrieben (siehe Kapitel 3.4). Die Beschreibung einer Grid-Anwendung besteht aus der GResourceDL-Beschreibung aller zu verwendenden Ressourcen sowie aus der Definition der Abhängigkeiten der Ressourcen untereinander. Die GJobDL wird von der Grid-Architektur benötigt, um die Grid-Anwendung auf das eigentliche Computing-Grid abzubilden und um den Prozessablauf der Anwendung zu steuern.

Der Schwerpunkt der GADL 0.2 liegt in der Spezifikation der GResourceDL und der GJobDL. Die GInterfaceDL und die GDataDL sind noch in einem „pre-alpha“-Stadium und werden zunächst von der Grid-Architektur des Fraunhofer Resource Grids nicht unterstützt.

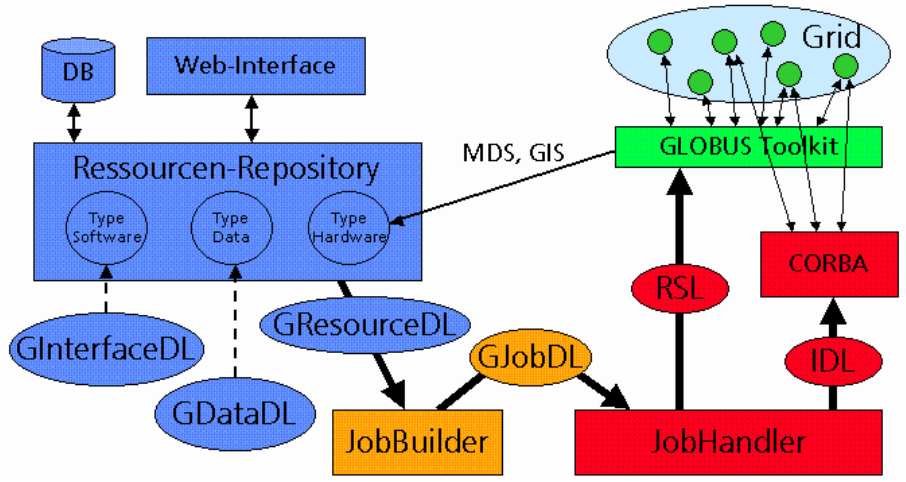

Abbildung 1: Die Beziehungen zwischen den Elementen der GADL

2 XML

Die GADL basiert auf den Spezifikationen der *Extensible Markup Language (XML)* [*Bray et al.*, 2000]. Die Verwendung von XML-basierten Beschreibungssprachen bietet einige Vorteile:

- Erweiterbarkeit: Die Semantik einer XML-Sprache kann individuell für jede Anwendung angepasst werden.
- Struktur: Die Struktur von XML-Dokumenten ist in einer externen *Document Type Definition (DTD)* oder in einem *XML Schema* festgelegt. Dadurch können Parser die Validität und Wohlgeformtheit von XML-Dokumenten überprüfen.
- Textformat: XML unterstützt alle gängigen Textkodierungen (ASCII, ISO8859-1, Unicode etc.) und ist daher unabhängig von der Plattform, der Programmiersprache und dem Protokoll, das für den Transport der XML-Dokumente verwendet wird.
- Dokumentation: XML-Dokumente enthalten Metadaten, die eine für Computer und Mensch verständliche Dokumentation der Daten ermöglichen.
- Verbreitung: XML ist weit verbreitet und es gibt viele Programme zum Bearbeiten, Speichern, Parsen, Umformen und Auswerten von XML-Dokumenten (siehe Kapitel 2.1). Viele der gängigen Beschreibungssprachen und Protokolle basieren auf XML. Als Beispiele sind WSDL [*Christensen et al.*, 2001] und SOAP [*Box et al.*, 2000; *Gudgin et al.*, 2001; *Apache*, 2001] zu nennen.

Die größten Nachteile, die sich durch die Verwendung von XML ergeben, sind das große Datenvolumen, das durch die zusätzlichen Metadaten und die Kodierung der Daten in einem Textformat verursacht wird, sowie der relativ rechenaufwändige Prozess des Parsens von XML-Daten. In unserem Fall ist jedoch die Zeitspanne, die für das Parsen und Übermittlen der GADL-Daten benötigt wird, in der Regel trotzdem vernachlässigbar gegenüber der Zeitspanne, die zum Beispiel für das Submitten und Ausführen einer Anwendung per Globus benötigt wird.

2.1 Werkzeuge

Im Folgenden sind einige Werkzeuge aufgezählt, die zur Entwicklung und Verarbeitung der GADL benutzt werden können.

- **TurboXML:** Kommerzielles Werkzeug von TIBCO zur Generierung von DTDs und Schemata sowie von XML-Dokumenten unter Windows und Linux/UNIX. Es gibt auch eine kostenlose 30 Tage Demo-Lizenz (http://www.tibco.com/solutions/products/extensibility/turbo_xml.jsp)
- **rxp:** Kleines Programm unter Linux zur Validierung von XML-Dokumenten (http://www.cogsci.ed.ac.uk/~richard/rxp.html)
- **tdtd:** DTD-Unterstützung für XEmacs: Debian-Paket tdtd
- **xerces2-j:** XML-Parser für Java (http://xml.apache.org/xerces2-j/index.html)
- **expat:** Streambasierter XML-Parser für C (http://sourceforge.net/projects/expat/)
- **xerces-c:** XML-Parser für C++ (http://xml.apache.org/xerces-c/index.html)

2.2 XML-Syntax

Die XML-Syntax ist in einer Empfehlung des World Wide Web Consortiums (W3C) festgehalten [*Bray et al.*, 2000]. Da Spezifikationen in der Regel sehr unübersichtlich sind, soll hier eine kurze Einführung in die XML-Syntax, angelehnt an *Anderson* [2000] gegeben werden.

Elemente sind die Grundbausteine eines XML-Dokuments. Ein Element ist eine Art Container für beliebige Inhalte. Ein Element kann beliebige Zeichen, andere Elemente und/oder andere Informationen enthalten. Elemente werden durch ein *Start-Tag* eingeleitet. Das Start-Tag besteht aus dem Namen des Elementtyps, der zwischen spitzen Klammern steht (z. B. `<resource>`). Das Ende eines Elements wird durch das *End-Tag* markiert. Das End-Tag besteht aus einem Schrägstrich, gefolgt von dem Namen des Elementtyps und das Ganze zwischen spitzen Klammern (z. B. `</resource>`). Jedes End-Tag muss mit einem Start-Tag korrespondieren. Leere Elemente haben keinerlei Inhalt und werden durch ein *Leeres-Element-Tag* markiert, das aus dem Namen des Elementtyps, gefolgt von einem Schrägstrich zwischen spitzen Klammern besteht (z. B. `<EOF/>`).

Jedes wohlgeformte XML-Dokument besteht aus einem einzigen Baum von Elementen. Dieser Baum hat nur eine Wurzel, die Dokumentwurzel (*document root*). An dieser Wurzel hängt ein Teilbaum von Elementen, dessen Wurzel das Dokument-Element (*document element*) ist. Bei der GResourceDL heißt das Dokument-Element zum Beispiel `<fhrgResources>`. Alle anderen Elemente in einem Dokument sind *Unterelemente* des Dokument-Elements. XML fordert strikt, dass Elemente korrekt geschachtelt sein müssen.

Text-Literale dienen als Werte für Attribute, interne Entities und externe Bezeichner. Text-Literale müssen von Begrenzern umschlossen sein. Begrenzer sind das Apostroph (') oder Anführungsstriche ("). *Character-Data* ist jeder Text, der nicht zum Markup gehört,

also der Inhalt von Elementen oder die Werte von Attributen. Zeichen wie `&` oder `<` dürfen nicht in Character-Data auftauchen, können aber durch Entities wie zum Beispiel `&` oder `<` ersetzt werden. Diese Entities werden dann vom XML-Parser aufgelöst.

Zu einem Element können mehrere *Attribute* angegeben werden. Jedes Attribut besteht aus einem Paar aus Attribut-Name und Attribut-Wert (z. B. `type="software"` oder `type='software'`). Innerhalb eines Start-Tags oder Leeres-Element-Tags ist jeweils nur eine Instanz eines Attribut-Namens erlaubt, z. B.:

```
<resource type="softwareClass" id="linux">
```

Verarbeitungsanweisungen werden in XML in der Form `<?Ziel ...Anweisungen... ?>` definiert. Die *XML-Deklaration* in der ersten Zeile eines jeden XML-Dokuments bedient sich einer ähnlichen Syntax:

```
<?xml version = "1.0" encoding ="ISO-8859-1"?>
```

In der GADL 0.2 wird von Verarbeitungsanweisungen kein Gebrauch gemacht.

Die grundlegende Syntax eines *Kommentars* in XML sieht so aus:

```
<!-- ...Kommentar... -->
```

Der Kommentar selber darf beliebige Zeichen außer der Zeichenfolge „`--`“ beinhalten und darf nicht mit einem „`-`“ enden.

Die Namen von Elementtypen und die Werte von Attributen werden bei der GADL analog zu den Namenskonventionen zu Variablennamen bei Java [*Sun Microsystems*, 1999] vergeben.

Attribut versus Element

Ursprünglich waren diese beiden Möglichkeiten bei SGML zur Unterscheidung von Inhalten (Unterelemente) und Metadaten (Attribute) gedacht. Bei der Übertragung von strukturierten Daten ist diese Unterscheidung hinfällig. Daher erfolgt hier die Abwägung, ob eine Information als Attribut oder als Inhalt eines Elements ausgedrückt werden soll, rein nach praktischen Gesichtspunkten [*Anderson*, 2000]:

- Vorteile von XML-Attributen
  - Einschränkung des gültigen Wertebereichs in der DTD möglich, besonders sinnvoll bei kleinen Wertebereichen
  - Definition eines Standardwertes in der DTD
  - ID und IDREF können überprüft werden
  - Geringerer Platzverbrauch als bei Elementen
  - Normalisierung von Leerstellen bei einigen Datenarten (NMTOKENS)
  - Einfache Verarbeitung durch DOM und SAX
  - Zugriff auf nicht (durch den Parser) analysierte externe Entities, zum Beispiel auf binäre Daten
- Nachteile von XML-Attributen
  - Nur Text als Wert zulässig, muss also beim Parsen gegebenenfalls in andere Typen umgewandelt werden
  - Keine Metadaten, also zum Beispiel Attribute von Attributen, erlaubt

  - Es kann keine Reihenfolge der Attribute vorgegeben werden

- Vorteile Unterelemente
  - Unterstützung von beliebig komplexen und wiederholenden Werten
  - Die Reihenfolge der Elemente kann vorgeben werden
  - Beliebige Metadaten
  - Einfacher erweiterbar, falls sich das Datenmodell ändert (Zukunftsfähigkeit)
- Nachteile Unterelemente
  - Höherer Platzverbrauch
  - Aufwändigere Programmierung; der Inhalt eines Elements kann bei SAX-Parsern auf mehrere Aufrufe der Methode characters() verteilt sein.

Insbesondere häufig auftretende Parameter sollten dementsprechend möglichst als Attribut ausgedrückt werden, um die XML-Dokumente kompakt zu halten und eine spätere Auswertung der XML-Daten zu beschleunigen.

2.3 Datenmodell in XML

Ein Datenmodell dient dazu, die Strukturen und Bedeutungen der Informationen, die in einem Dokument gespeichert werden sollen, zu erfassen, um sie später durch Regeln innerhalb einer DTD oder eines Schemas umzusetzen [*Anderson*, 2000]. In der GADL spielen dabei sowohl statische Modelle, die die zulässigen Zustände eines Systems beschreiben, als auch dynamische Modelle, die den Datenfluss im System beschreiben, eine Rolle.

Die GADL 0.2 wird an Hand von vier DTDs definiert, die jeweils das Datenmodell und das Vokabular der GResourceDL, GInterfaceDL, GDataDL und GJobDL festlegen. Diese DTDs können jederzeit in Schemata konvertiert werden. Da der JobBuilder des Fraunhofer Resource Grids zur Zeit jedoch auf Grund der verwendeten Parsertechnologie (JAXB) noch keine Schemata verarbeiten kann, wird der Umstieg von DTDs auf Schemata erst in einer späteren Version der GADL erfolgen.

Die DTDs werden einerseits auf dem projektinternen BSCW-Server abgelegt, zum anderen sind sie unter der URL http://www.fhrg.fhg.de/de/fhrg/*.dtd online erreichbar. Der Verweis auf eine URL kann notwendig sein, wenn man XML-Dokumente über Sockets überträgt, da man sonst Probleme mit relativen Pfadangaben einer lokalen DTD bekommen kann. Die Angabe der DTD in einem XML-Dokument könnte also zum Beispiel wie folgt aussehen:

```
<?xml version = "1.0" encoding = "ISO-8859-1"?>
<!DOCTYPE fhrgResources SYSTEM "http://www.fhrg.fhg.de/de/fhrg/grdl0_2.dtd">
<fhrgResources>
 ...
</fhrgResources>
```

3 Grid Application Definition Language (GADL)

Bei dem vorliegenden Entwurf der GADL wurde versucht, das jeweilige Datenmodell möglichst generisch, übersichtlich und eindeutig zu gestalten. Da ausgeprägte Hierarchien in der Regel zu größeren XML-Dokumenten führen, soll besonders bei häufig auftretenden Parametern Wert auf flache Hierarchien gelegt werden. Die GADL soll dabei möglichst unabhängig von der konkreten Grid-Architektur und frei konfigurierbar bleiben.

Im Vergleich zu vielen anderen Beschreibungssprachen für Grid-Ressourcen und Grid-Jobs wird bei der GADL ein höherer Abstraktionsgrad angestrebt. Abbildung 2 zeigt zum Beispiel ein Modell einer Beschreibungssprache, die vom NCSA zur Beschreibung von Grid-Jobs entwickelt wurde (http://portals.ncsa.uiuc.edu/schemas/). Hier wird explizit Bezug auf die RSL genommen; diese Beschreibung ist somit nicht unabhängig von der Grid-Architektur.

Das *Texas Advanced Computing Center (TACC)* entwickelt zur Zeit eine Beschreibungssprache für Hardwareressourcen, die für ein *Portal Grid Information System (PIS)* verwendet werden soll (http://www.tacc.utexas.edu/ ~rich/portal_info_services/). Die Abbildungen 3 bis 6 zeigen die XML-Modelle der jeweiligen Elemente mit deren Unterelementen. In dieser Beschreibungssprache wird jede Art von Ressource getrennt behandelt und jeweils auf ein spezifisches XML-Element abgebildet. Das hat den Nachteil, dass das Datenmodell sehr umfangreich ist und für jeden hinzukommenden Ressourcentyp erweitert werden muss. Bei der GADL soll daher ein rekursiver Ansatz gewählt werden, bei dem alle Arten von Ressourcen mit dem selben Datenmodell beschrieben werden. In der GADL werden Elemente wie zum Beispiel `<computeResource>`, `<node>` oder `<processor>` alle als Element `<resource>` behandelt und miteinander verknüpft (siehe auch Kapitel 3.1).

Wird Globus als zugrundeliegende Grid-Middleware verwendet, so muss für die konkrete Ausführung die Anwendung in der *Globus Resource Specification Language (RSL)* beschrieben werden [*The Globus Project*, 2000*b*,*a*]. Ein Modell der im *Java Commodity Grid Kit (Java-CoG)* [*von Laszewski et al.*, 2001] verwendeten XML-Syntax der RSL ist in Abbildung 7 dargestellt. Die Semantik der XML-RSL ist für die abstrakte Beschreibung von komplexen Grid-Anwendungen jedoch viel zu allgemein gehalten, da es kaum vordefinierte Elemente gibt. Zudem lässt sich die XML-RSL nur sehr schlecht kategorisieren.

Ein weiterer Ansatz zur Beschreibung von Hardwareressourcen, dessen Syntax zum Teil in die GADL 0.2 eingeflossen ist, ist in *von Laszewski* [2002] zu finden.

Im Folgenden werden die einzelnen GADL-Bausteine *GResourceDL, GInterfaceDL, GDataDL* und *GJobDL* an Hand ihres Datenmodells und einiger Beispiele vorgestellt und erläutert.

3.1 Grid Resource Definition Language (GResourceDL)

Die GResourceDL dient zur Beschreibung der Ressourcen eines Computing-Grids nach dem Motto „alles ist eine Ressource“. In der GADL 0.2 wird die Beschreibung folgender Ressourcentypen unterstützt:

- konkrete Softwarekomponenten (`type="software"`),
- Softwareklassen (`type="softwareClass"`),
- konkrete Hardwareressourcen (`type="hardware"`),

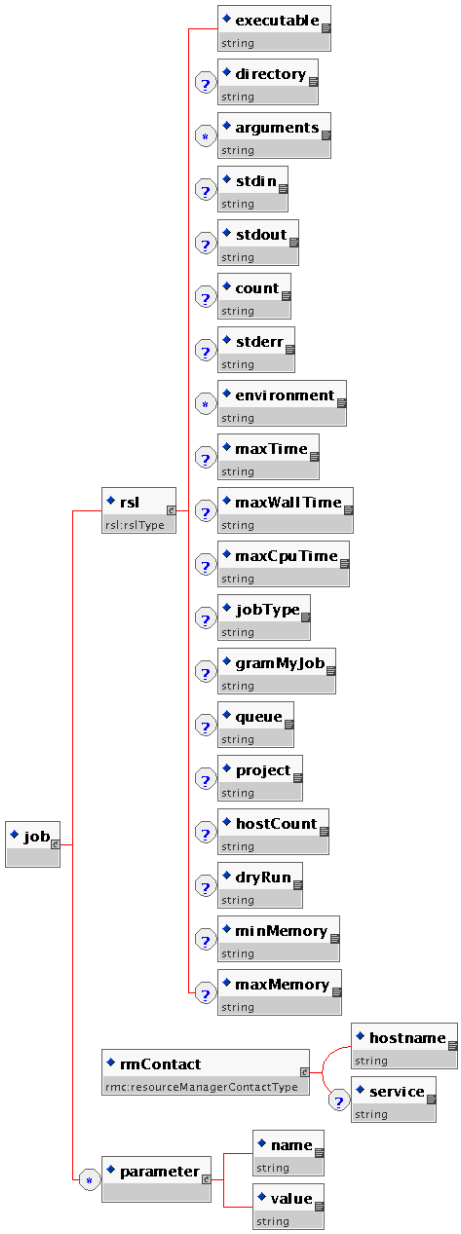

Abbildung 2: Eine vom NCSA entwickelte Beschreibungssprache für Grid-Jobs (http://portals.ncsa.uiuc.edu/schemas/)

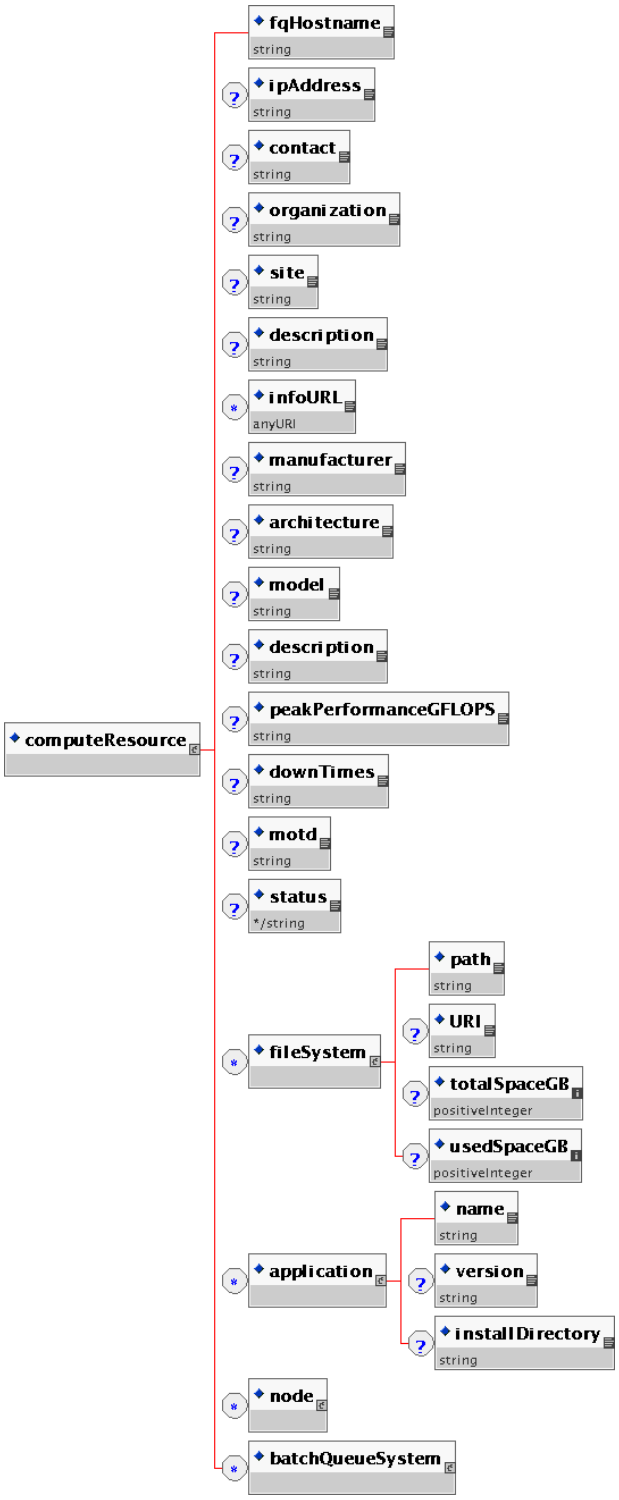

Abbildung 3: Das Element *computeResource* des *Portal Grid Information Systems* vom TACC.

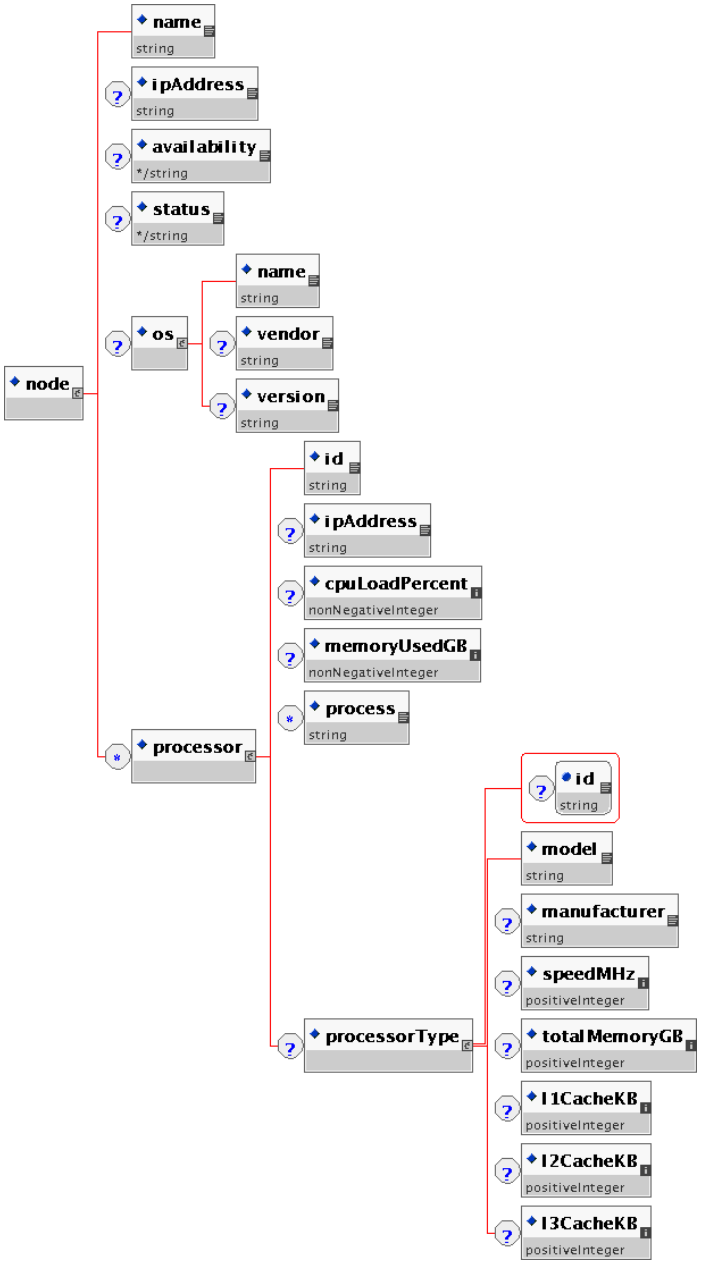

Abbildung 4: Das Element *node* des *Portal Grid Information Systems* vom TACC.

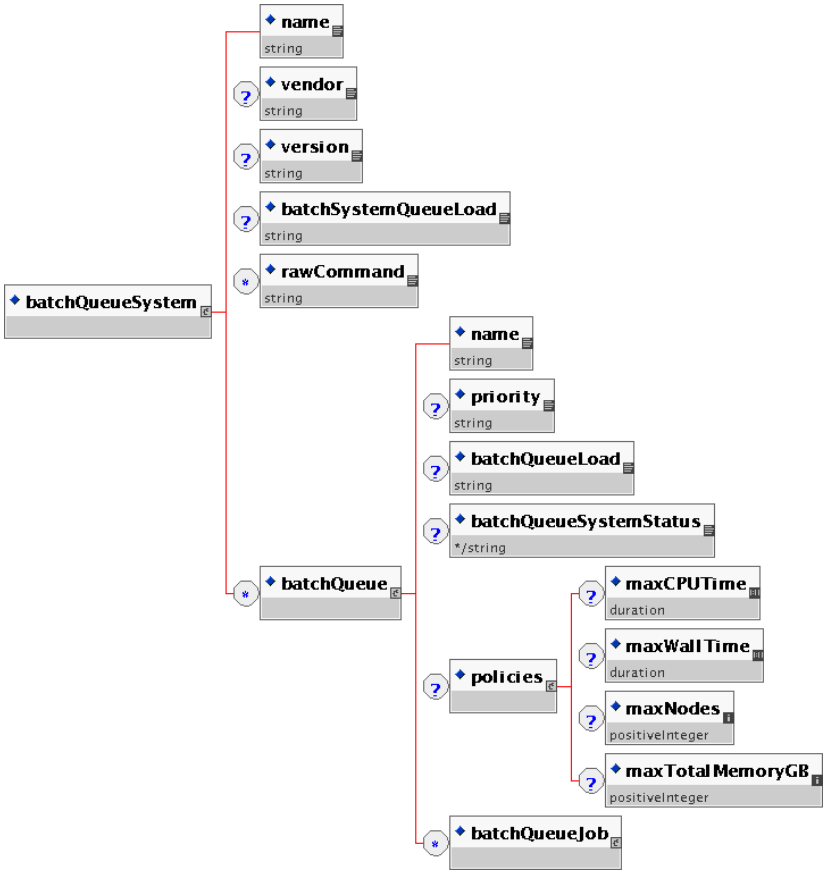

Abbildung 5: Das Element *batchQueueSystem* des *Portal Grid Information Systems* vom TACC.

- Hardwareklassen (`type="hardwareClass"`),
- konkrete Daten (`type="data"`) und
- Datenklassen (`type="dataClass"`).

Es können mit der GResourceDL also sowohl konkrete Instanzen als auch bestimmte Klassen von Objekten beschrieben werden. Als Instanz bezeichne ich hier Objekte, die tatsächlich physisch vorhanden sind, also eine eindeutige Ortsangabe besitzen. Diese Ortsangabe kann zum Beispiel durch die IP-Nummer einer konkreten Hardware(-instanz) oder durch die URI einer Datei ausgedrückt werden. Eine Klasse von Objekten hat hingegen keine eindeutige Ortsangabe. Eine Hardwareklasse wäre zum Beispiel eine bestimmte Art von Prozessor (AMD 1000MHz) oder die Angabe einer minimalen Speichergröße (RAM $>$ 512MB). Als Beispiel für eine Softwareklasse ist die Ressource „Linux“ zu nennen.

Im Gegensatz zu anderen Beschreibungssprachen werden in der GResourceDL hierarchisch aufgebaute Hardwareressourcen im Allgemeinen rekursiv beschrieben, was ich an einem kurzen Beispiel erläutern möchte: Angenommen, wir haben einen Linux-Cluster mit 16 Knoten mit je 2 Prozessoren. Nun könnte man diesen Cluster analog zur Vorgehensweise des TACC (siehe Abbildungen 3 bis 6) so beschreiben, dass man für jede Art von

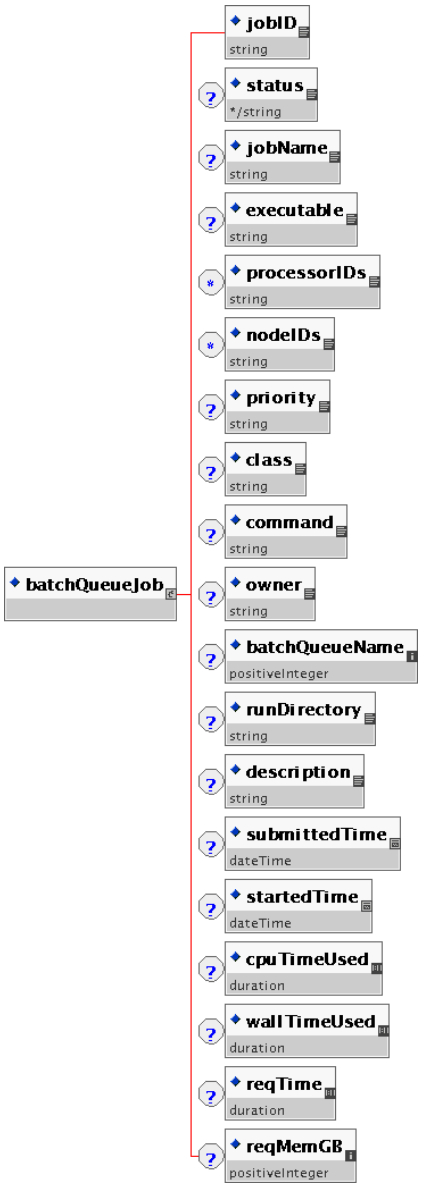

Abbildung 6: Das Element *batchQueueJob* des *Portal Grid Information Systems* vom TACC.

Abbildung 7: Eine XML-Version der *Resource Specification Language (RSL)*, die für das Ausführen von Anwendungen per *Java Commodity Grid Kit* verwendet wird.

Hardware eine eigene Elementbezeichnung einführt und diese dann hierarchisch anordnet (computeResource → node → processor). In der GResourceDL würde man hingegen zunächst drei Hardwareklassen mit der entsprechenden Beschreibung definieren. Die konkreten Instanzen (1 x cluster, 16 x node, 32 x processor) können nun die Eigenschaften dieser Hardwareklassen mit Hilfe des Konstrukts `<dependencies type="provides">` erben. Die Verknüpfung der einzelnen Instanzen erfolgt ebenso über die *provides*-Beziehung. Diese rekursive Art der Beschreibung ermöglicht es, mit relativ wenig Sprachelementen beliebig tief geschachtelte Ressourcen zu erfassen. Die Ausnahme bilden Informationen, die sehr oft gebraucht werden (Speicherplatz, Dateisysteme etc.). Für diese Elemente sind aus Gründen der Performance eigene Elementtypen vorgesehen, die fest in der XML-Spezifikation der GResourceDL verankert sind.

3.1.1 Ressourcen-Mapping

Die Kategorisierung und Beschreibung der Ressourcen eines Computing-Grids dient in erster Linie dazu, ein so genanntes *Ressourcen-Mapping* zu ermöglichen, durch das eine Problemstellung auf die hierfür geeigneten konkreten Software- und Hardwarekomponenten sowie Daten abgebildet wird. Die Grid-Architektur des Fraunhofer Resource Grids verwendet hierfür ein mehrstufiges Verfahren:

1. **Task-Mapping:** Mit Hilfe einer Task Mapping Engine (TME) wird zunächst die Problemstellung interaktiv vom Benutzer auf vorhandene Softwarekomponenten oder Softwareklassen abgebildet.

2. **JobBuilder:** Die durch das Task-Mapping bestimmte Menge von Softwarekomponenten und -klassen wird anschließend nach den Wünschen des Benutzers mit Hilfe des JobBuilders zu einer komplexen Anwendung zusammengesetzt. Die explizit zu verwendenden Hardwareressourcen (Hardwareklassen oder konkrete Hardware) können bei Bedarf durch den Benutzer vorgewählt werden. Abhängigkeiten und Konflikte zwischen den ausgewählten Ressourcen werden durch den JobBuilder angezeigt. Der Prozessablauf der komplexen Anwendung wird in Form eines Petrinetzes festgelegt.

3. **JobHandler:** Der JobHandler erhält vom JobBuilder eine Auswahl von Ressourcen, die durch unterschiedliche Arten von Abhängigkeiten miteinander verknüpft sind. Der JobHandler löst nun die vom JobBuilder noch nicht aufgelösten Abhängigkeiten auf, indem er zu jeder Ressource mit einer zwingenden Anforderung (`<dependencies type="depends">`) alle konkreten Ressourcen sucht, die diese Anforderung direkt oder indirekt erfüllen (`<dependecies type="provides">`) und die dabei nicht in Konflikt zu anderen verwendeten Ressourcen stehen. Diese Suche kann zum Beispiel mittels automatischer Generierung von XPath- bzw. XQuery-Abfragen erfolgen. Das Auflösen der Abhängigkeiten ist ein iterativer Prozess, bei dem gegebenenfalls über mehrere Zwischenstufen von Ressourcen iteriert wird, bevor konkrete Ressourcen gefunden werden, die alle Anforderungen erfüllen und keine weiteren Abhängigkeiten nach sich ziehen.

   Im Allgemeinen wird es zu einer Softwarekomponente mehrere konkrete Hardwareressourcen geben, auf denen sie potentiell ausgeführt werden kann.

4. **Meta-Scheduler:** Nun ist es Aufgabe des Meta-Schedulers, aus der vom JobHandler ausgewählten Menge möglicher Hardwareressourcen die (gemäß festzulegender Kriterien) geeignetste herauszusuchen. Die Entscheidung des Meta-Schedulers hängt dabei vom aktuellen Zustand des Computing-Grids ab. Folgende Elemente der GResourceDL könnten zum Beispiel Einfluss auf die Wahl der geeignetsten Hardwareressource haben:

   - `<cost>`: Kosten, die bei Benutzung dieser Ressource anfallen.
   - `<benchmark.hardware>`: Leistungsfähigkeit der Hardwareressource bezüglich bestimmter Benchmarks
   - `<transmissionSpeed>`: Übertragungsgeschwindigkeit zum Beispiel einer Netzwerkkarte
   - `<cpu.loadAverage>`: Auslastung der CPU in vergangener Zeit
   - `<benchmark.software>`: Leistungsbedarf einer Software bezüglich bestimmter Benchmarks

   Zusätzlich zu den Einträgen in der GResourceDL kann es sinnvoll sein, wenn sich der Meta-Scheduler über den Zugriff auf lokale Scheduler oder durch die Abfrage eines Grid-Informationssystems (GIS, MDS, Ressourcen-Repository etc.) ein Bild vom aktuellen Zustand der verfügbaren Hardwareressourcen macht.

5. **Batch/Queue-Systeme bzw. lokale Scheduler:** Die Zuordnung der Softwarekomponente mit der vom Meta-Scheduler ausgewählten Hardwareressource geschieht mit Hilfe des Elements `<executionLocation>`. Der JobHandler wertet dieses Element aus, um die Softwarekomponente per GRAM auf der zugeordneten Hardwareressource auszuführen. In dem Fall, dass der mittels GRAM angesprochene Globus-Jobmanager als Gatekeeper mehrerer, zu einem Cluster zusammengefassten Rechnerknoten fungiert, übernimmt das lokale *Batch/Queue-System* (oft auch *lokaler Scheduler* genannt) die endgültige Zuordnung der Softwarekomponente auf einen oder mehrere freie Knoten des Clusters.

Damit das Ressourcen-Mapping in dieser Form praktikabel bleibt, muss ein verbindlicher Grundstock an standardisierten Ressourcenklassen (z. B. linux, x86, globus-node, network-card etc.) vordefiniert werden. Die Beschreibung der Anforderungen neuer Ressourcen sollte sich soweit wie möglich an diesen Grundstock halten. Durch ein geeignetes Werkzeug muss beim Einpflegen neuer Ressourcen sichergestellt werden, dass nur auf bereits vorhandene Ressourcen-IDs verwiesen wird, da sonst unnötige Laufzeitfehler beim Auflösen der Abhängigkeiten entstehen.

Bei der Konzeption der GResourceDL wurde versucht, eine möglichst allgemeingültige Syntax zu finden, die prinzipiell unabhängig vom Fraunhofer Resource Grid ist. Ziel ist dabei, dass die GResourceDL für verschiedene Systeme (mit unterschiedlichen JobHandlern und Schedulern) frei konfigurierbar bleibt. Von den Benutzern sollen beliebige Abhängigkeiten definiert werden können, die dann von dem Ressourcen-Mapping auf generische Art und Weise aufgelöst werden.

In dieser Version der GResourceDL wird davon ausgegangen, dass die zu beschreibenden Softwarekomponenten ausschließlich auf Hardwareressourcen auszuführen sind, die

eine standardisierte Schnittstelle zum Ausführen und zum Monitoring der Softwarekomponente (z. B. per Globus) bieten. Die Beschreibung von Anforderungen an Hardwareressourcen außerhalb des Computing-Grids (z. B. Rechner externer Nutzer) ist bisher nicht getestet worden; vermutlich fehlen hierfür noch spezielle Sprachelemente. Dateien oder Anwendungen, die sich zum Beispiel auf dem Laptop eines externen Nutzers befinden, lassen sich somit bisher nicht ohne weiteres in eine Grid-Anwendung integrieren, ohne sie zuvor (z. B. per FTP) auf einen Grid-Rechner zu transferieren.

Man kann sich das Ressourcen-Mapping mit Hilfe des Elementtyps `dependencies` wie eine Börse vorstellen, auf der zwischen Angebot (`provides`) und Nachfrage (`depends`) von Ressourcen vermittelt wird. In Grid-Architekturen nennt man daher den Service, der für das Ressourcen-Mapping zuständig ist, oft auch *Broker*.

3.1.2 Datenmodell

Im Folgenden werden das Datenmodell der GResourceDL vorgestellt und die wichtigsten Elemente erläutert. Eine Online-Dokumentation der GResourceDL ist unter
http://www.fhrg.fhg.de/de/fhrg/GADL_0_2/docs/grdl0_2_dtd/
zu finden. Nach dem Namen des Elementtyps mit den dazugehörenden Attributen in spitzen Klammern wird jeweils das Inhaltsmodell des Elements in runden Klammern gemäß der DTD-Syntax angegeben:

- `(#PCDATA)`: Parsed Character DATA: Inhalt wird unausgewertet weitergegeben
- `EMPTY`: Element vom Typ Leeres-Element
- `ANY`: Dieses Element kann beliebigen Inhalt haben
- `(element*)`: Element kommt null oder mehrmals vor
- `(element+)`: Element kommt ein- oder mehrmals vor
- `(element?)`: Element ist optional (null oder einmal)
- `(element1|element2)`: Entweder element1 oder element2
- `(element1,element2)`: Beide Elemente müssen in dieser Reihenfolge vorkommen

---

`<fhrgResources>`

`(resource*)`

Dieses Element ist das Dokument-Element der GResourceDL und muss damit genau einmal in dem Dokument vorkommen (Abbildung 8). Der Inhalt von Element `<fhrgResources>` kann aus null oder mehreren Elementen vom Typ `<resource>` bestehen.

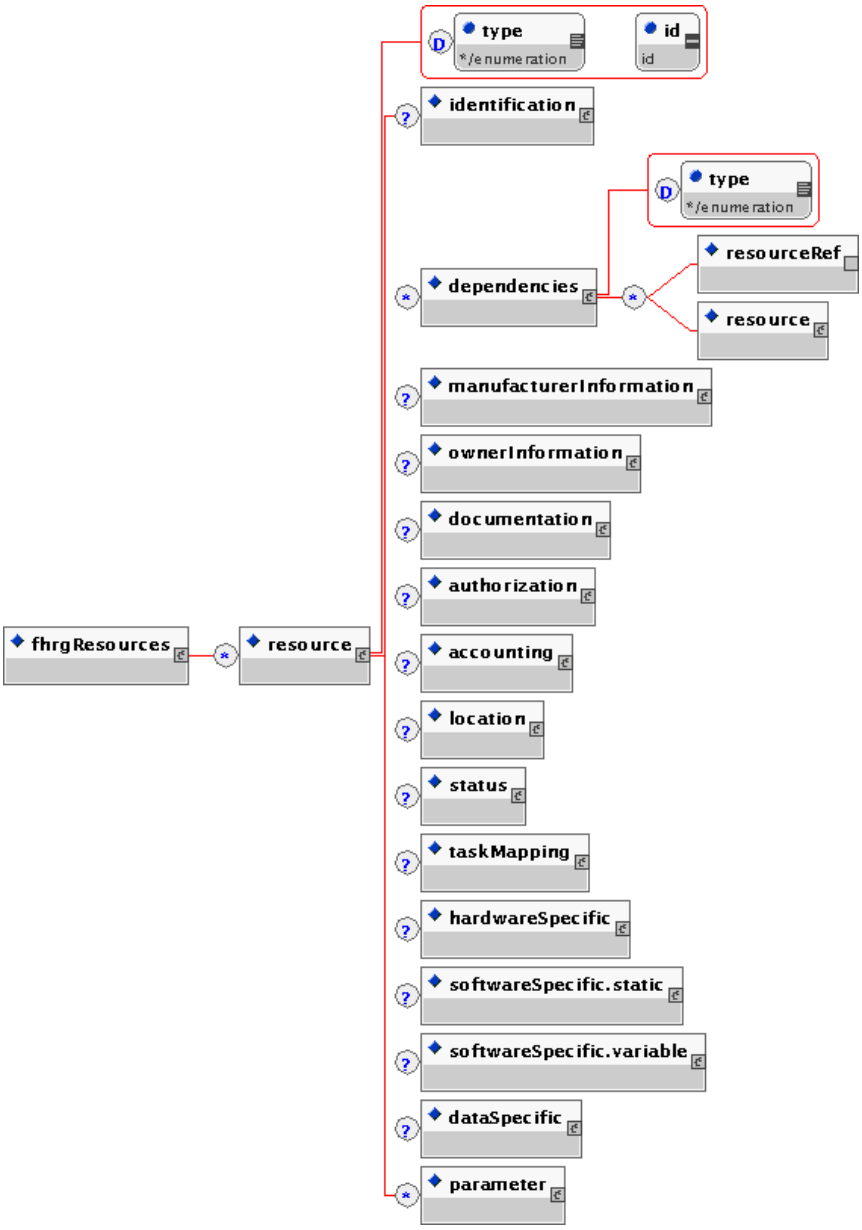

Abbildung 8: Der Elementtyp *fhrgResources* der GResourceDL mit einigen Unterelementen.

```
<resource id="..." type="...">

(identification? , dependencies* , manufacturerInformation? ,
 ownerInformation? , documentation? , authorization? ,
 accounting? , location? , status? , taskMapping? ,
 hardwareSpecific? , softwareSpecific.static? ,
 softwareSpecific.variable? , dataSpecific? , parameter*)
```

Mit Hilfe des Attributs `id` erfolgt eine eindeutige Kennzeichnung (Primärschlüssel ohne Wiederholung) der Ressourcenbeschreibung für das gesamte Computing-Grid. Für jede Kennung gibt es genau eine Ressourcenbeschreibung. Verschiedene Versionen einer Ressource erhalten verschiedene Kennungen und damit auch verschiedene Ressourcenbeschreibungen. Ich schlage folgende Syntax für die Kennung vor:

`TTDD_NNNNNN_{reverseDomain}_{resourceName}`

`TT`: Kürzel für Ressourcentyp ($TT \in \{SW, SC, HW, HC, DA, DC\}$)

SW	konkrete Softwarekomponente
SC	Softwareklasse
HW	konkrete Hardwareinstanz
HC	Hardwareklasse
DA	konkrete Daten
DC	Datenklasse

`DD`: Domäne bzw. Namespace, in dem die Kennung gültig ist

00	Gesamtes Fraunhofer Resource Grid
01	Fraunhofer ITWM
02	Fraunhofer FIRST
03	Fraunhofer IAO
04	Fraunhofer IGD
05	Fraunhofer SIT

`NNNNNN`: Durchlaufende Ressourcen-Nummer, die für den angegebenen Ressourcentyp und die Domäne bzw. den Namespace eindeutig ist. Wird einfach für jeden Ressourcentyp und jeden Namespace getrennt hochgezählt.

`{reverseDomain}`: Textrepräsentation von `DD` in der Form, wie sie auch bei Java-Paketnamen üblich ist. Als Trennzeichen ist „-“ zu verwenden.

de-fhrg	Gesamtes Fraunhofer Resource Grid
de-fhrg-itwm	Fraunhofer ITWM
de-fhrg-first	Fraunhofer FIRST
de-fhrg-iao	Fraunhofer IAO
de-fhrg-igd	Fraunhofer IGD
de-fhrg-sit	Fraunhofer SIT

`{resourceName}`: Ressourcenname, der analog zu `NNNNNN` für den angegebenen Ressourcentyp und den Namespace eindeutig ist.

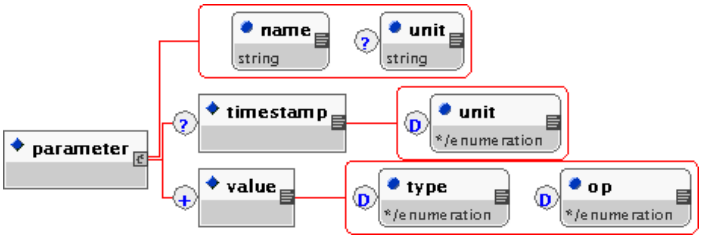

Abbildung 9: Der Elementtyp *parameter* der GResourceDL mit einigen Unterelementen.

Mögliche Ressourcenkennungen sind zum Beispiel:

- `HC00_000001_de-fhrg_x86`: `HC`=Hardwareklasse, `00`=Namespace FhRG, `000001`=Ressourcen-Nummer, `de-fhrg`=Namespace FhRG, `x86`=Ressourcenname.
- `HW02_000001_de-fhrg-first_harlekin`: `HW`=Hardware-Instanz, `02`=Namespace FIRST, `000001`=Ressourcen-Nummer, `de-fhrg-first`=Namespace FIRST, `harlekin`=Ressourcenname.
- `SC00_000003_de-fhrg_glibc6`: `SC`=Softwareklasse, `00`=Namespace FhRG, `000003`=Ressourcen-Nummer, `de-fhrg`=Namespace FhRG, `glibc6`=Ressourcenname.

Das Attribut `type` soll die Suche nach bestimmten Typen von Ressourcen erleichtern. Zur Zeit unterstützte Typen sind:

`software`	Softwarekomponente, die tatsächlich physisch vorhanden ist, zum Beispiel als eine ausführbare Datei (z. B. harlekin.first.fraunhofer.de:/usr/bin/sleep)
`softwareClass`	Softwareklasse (z. B. Linux)
`hardware`	Hardwareressource, die tatsächlich physisch vorhanden ist und die im Computing-Grid lokalisiert werden kann
`hardwareClass`	Hardwareklasse (z. B. x86)
`data`	Daten, die tatsächlich physisch vorhanden sind (z. B. als Datei)
`dataClass`	Datenklasse (z. B. Wetterdaten)

```
<parameter name="..." unit="...">

(timestamp? , value+)
```

Dieser generische Elementtyp kann dazu benutzt werden, Eigenschaften einer Ressource zu spezifizieren, die nicht explizit als Elementtyp in der DTD definiert wurden (Abbildung 9). Mit dem Attribut `name` wird der Name der Eigenschaft angegeben, mit `unit` die Einheit, in welcher der Wert angegeben wird. Mittels des optionalen Unterelements `timestamp` kann ein Zeitpunkt angegeben werden, zu dem der Wert gültig war. Die Angabe des Wertes selber erfolgt über den Elementtyp `value`.

```
<timestamp unit="...">
(#PCDATA)
```

Mit `timestamp` lässt sich ein Zeitpunkt angeben (Abbildung 9). Die Einheiten sind je nach Attribut `unit` entweder `msSince1970` (Millisekunden seit 1.1.1970, 00:00, vgl. Java) oder `iso8601` (Zeitformat nach ISO 8601: YYYY-MM-DDThh:mm:ss.sss, z. B. 2000-01-01T08:12:00.000+02:00 oder 2000-01-12T12:13:14Z).

```
<value type="..." op="...">
(#PCDATA)
```

Eigenschaften von Ressourcen, die z. B. durch Zahlenwerte auszudrücken sind und die zu Vergleichen herangezogen werden sollen, können durch den Elementtyp `value` angegeben werden (Abbildung 9). Da DTDs keine direkte Formatspezifizierung des Elementinhalts ermöglichen, können mittels des Attributs `type` folgende Formate vorgegeben werden:

`float`	Textrepräsentation von IEEE-754 Fließkommazahlen mit einfacher Genauigkeit (4Byte), z. B. 3.14, 2f, 1e1, .5f, 6., 0, -0, INF, -INF, NaN
`double`	Textrepräsentation von IEEE-754 Fließkommazahlen mit doppelter Genauigkeit (8Byte)
`int`	Textrepräsentation von ganzen Zahlen (4Byte)
`boolean`	`true` oder `false`
`string`	Endliche Zeichenkette (Standardwert)
`base64`	base64-kodierte binäre Daten gemäß *Freed und Borenstein* [1996]

Das Attribut `op` steht für Operator. Unterstützte Operatoren sind:

`eq`	$=$ (gleich) (Standardwert)
`gt`	$>$ (größer als)
`ge`	$\geq$ (größer gleich)
`lt`	$<$ (kleiner als)
`le`	$\leq$ (kleiner gleich)

```
<identification>
(description , keyword* , icon* , parameter*)
```

Dieser Elementtyp enthält Schlüsselwörter und Beschreibungen, die zur Identifikation der Ressource dienen (Abbildung 10). Zudem kann der Ort eines Icons angegeben werden, durch das die Ressource bei der Visualisierung (z. B. im JobBuilder) repräsentiert wird. Mit Hilfe des Elementtyps `parameter` können weitere Eigenschaften dieser Ressource gesetzt werden, die noch nicht als Elementtyp in die DTD aufgenommen wurden. Für den Elementtyp `keyword` sollte noch ein verbindlicher Grundstock von Schlüsselwörtern festgelegt werden, um eine sinnvolle Kategorisierung sicherstellen zu können.

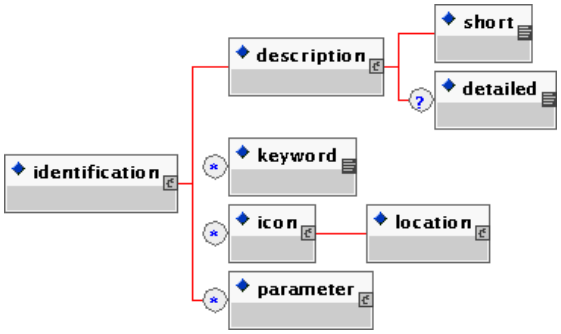

Abbildung 10: Der Elementtyp *identification* der GResourceDL mit einigen Unterelementen.

```
<dependencies type="...">

(resourceRef | resource)*
```

Mit Hilfe des Elementtyps `dependencies` werden alle wesentlichen Abhängigkeiten zwischen den Ressourcen definiert (Abbildung 8). Dabei kann entweder auf eine schon vorhandene Ressourcenbeschreibung mit Hilfe des Elements `resourceRef` verwiesen werden, oder es wird lokal eine neue (private) Ressourcenbeschreibung mit Hilfe des Elementtyps `resource` angelegt. Wird mittels einer Referenz auf eine existierende Ressourcenbeschreibung verwiesen, so muss die ID der Referenz mit der ID der gewünschten Ressource genau übereinstimmen. Die ID einer lokalen (privaten) Ressourcenbeschreibung muss nur innerhalb der übergeordneten Ressourcenbeschreibung eindeutig sein. Das Attribut `type` gibt an, um welche Art von Abhängigkeit es sich handelt:

`depends`	diese Ressource hängt zwingend von der genannten Ressource ab
`conflicts`	diese Ressource darf nicht zusammen mit der genannten Ressource verwendet werden
`provides`	diese Ressource stellt die genannte Ressource zur Verfügung oder enthält sie. Das bedeutet, dass sie deren Eigenschaften erbt und gegebenenfalls erweitert
`suggest`	es wird empfohlen, diese Ressource zusammen mit der genannten Ressource zu verwenden

Das Verfahren, wie der Elementtyp `dependencies` verwendet wird, um ein Ressourcen-Mapping durchzuführen, ist in Kapitel 3.1.1 näher erläutert. Hier ein kurzes Beispiel, wie die Vererbung von Eigenschaften zu verstehen ist:

```
<resource type = "softwareClass" id = "linux">
  <identification>
    <description>
      <short>linux operating system</short>
    </description>
    <keyword>linux</keyword>
    <keyword>operating system</keyword>
```

```
  </identification>
</resource>

<resource type = "softwareClass" id = "SuSE-Linux">
  ...
  <dependencies type="provides">
    <resourceRef type="softwareClass" id="linux"/>
  </dependencies>
</resource>

<resource type = "softwareClass" id = "Debian-Linux">
  ...
  <dependencies type="provides">
    <resourceRef type="softwareClass" id="linux"/>
  </dependencies>
</resource>
```

Hier werden die Eigenschaften der Ressource *linux* an die Ressourcen *SuSE-Linux* und *Debian-Linux* vererbt. Die IDs der in den Beispielen verwendeten Ressourcen entsprechen der Einfachheit halber nicht den vorgeschlagenen Konventionen.

Im nächsten Beispiel kann ein Softwarepaket nur auf bestimmten Rechnern ausgeführt werden, da sonst der Lizenzserver nicht erreicht wird, oder der Lizenzserver nicht zulässt, dass das Programm irgendwo anders läuft. Eine zeitliche Einschränkung, die z. B. besagt, dass ein Softwarepaket nur dann ausgeführt werden darf, wenn mindestens eine Lizenz frei ist, müsste mit dem Petrinetz der GJobDL-Spezifikation (siehe Kapitel 3.4) modelliert werden. (Die Transition *Softwarepaket* schaltet erst, nachdem die Transition *Lizenzserver* geschaltet hat und eine Marke in die Stelle *Lizenz frei* gelegt hat). Hierzu müsste der Lizenzserver aber wohl gekapselt werden, damit ein standardisierter Zugriff möglich ist.

```
<resource type = "software" id = "Softwarepaket15">
  ...
  <dependencies type = "depends">
    <resourceRef type = "softwareClass" id = "Lizenzserver"/>
  </dependencies>
</resource>

<resource type = "softwareClass" id = "Lizenzserver">
  <identification>
    <description>
      <short>Hardware that provides this resource has access to the Lizenzserver</short>
    </description>
  </identification>
</resource>

<resource type = "software" id = "Lizenzserver_at_p1">
  ...
  <dependencies type = "provides">
    <resourceRef type = "softwareClass" id = "Lizenzserver"/>
  </dependencies>
</resource>

<resource type = "hardware" id = "p1.itwm.uni-kl.de">
  ...
  <dependencies type = "provides">
```

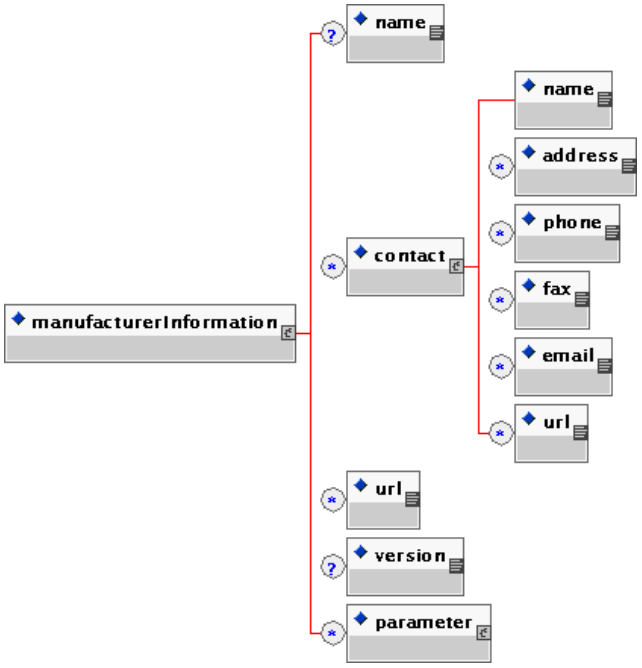

Abbildung 11: Der Elementtyp *manufacturerInformation* der GResourceDL mit einigen Unterelementen.

```
    <resourceRef type = "software" id = "Lizenzserver_at_p1"/>
  </dependencies>
</resource>

<resource type = "hardware" id = "harlekin.first.gmd.de">
  ...
  <dependencies type = "provides">
    <resourceRef type = "softwareClass" id = "Lizenzserver"/>
  </dependencies>
</resource>
```

Diese Abhängigkeiten lassen sich so lesen: Die Softwarekomponente *Softwarepaket15* benötigt Softwareklasse *Lizenzserver*. Diese Softwareklasse wird von Softwarekomponente *Lizenzserver_at_p1* und von der Hardware *harlekin.first.gmd.de* bereitgestellt. Die Softwarekomponente *Lizenzserver_at_p1* wird wiederum von Hardware *p1.itwm.uni-kl.de* bereitgestellt. Das *Softwarepaket15* kann somit auf *harlekin.first.gmd.de* oder auf *p1.itwm.uni-kl.de* ausgeführt werden.

`<manufacturerInformation>`

`(name? , contact* , url* , version? , parameter*)`

Mit Hilfe dieses Elementtyps können Informationen über den Hersteller der Ressource definiert werden (Abbildung 11). Darunter fällt auch die Versionsnummer der Ressource, da diese in der Regel durch den Hersteller vergeben wird. Desweiteren können mehrere

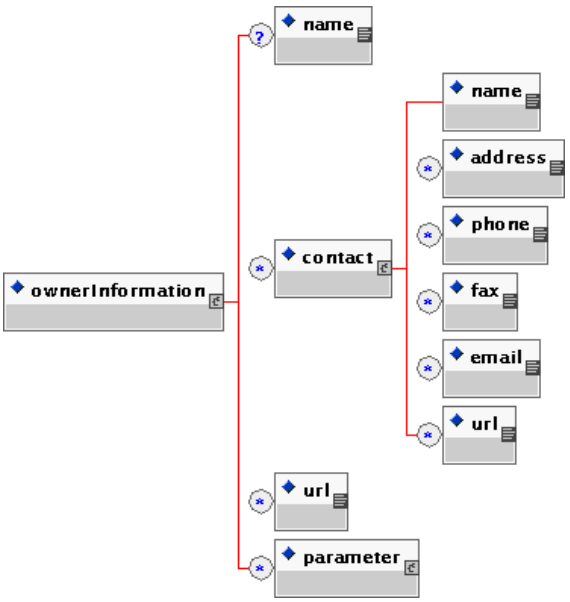

Abbildung 12: Der Elementtyp *ownerInformation* der GResourceDL mit einigen Unterelementen.

Kontaktadressen, URLs zu den Internetseiten des Herstellers sowie sonstige Parameter angegeben werden.

```
<ownerInformation>

(name? , contact* , url* , parameter*)
```

Analog zu `manufacturerInformation` können mit diesem Elementtyp Informationen über den Besitzer dieser Ressource abgelegt werden (Abbildung 12). Dies kann wichtig für das *Accounting* und *Billing* der Ressource werden.

```
<documentation>

(documentationRef* , thirdPartyDocumentation* , addedValue? ,
 exampleScenario* , parameter*)
```

Mittels `documentation` lässt sich angeben, wo Dokumentation zu dieser Ressource zu finden ist (Abbildung 13). Dokumentation, die nicht vom Hersteller herausgegeben wurde, ist unter dem Elementtyp `thirdPartyDocumentation` zu führen. `addedValue` gibt den Mehrwert an, der durch die Verwendung der Ressource erzeugt werden kann.

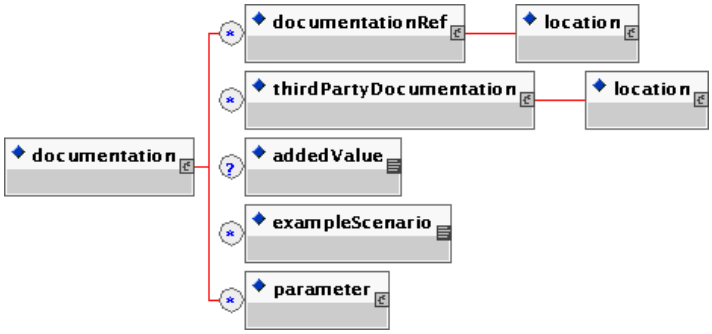

Abbildung 13: Der Elementtyp *documentation* der GResourceDL mit einigen Unterelementen.

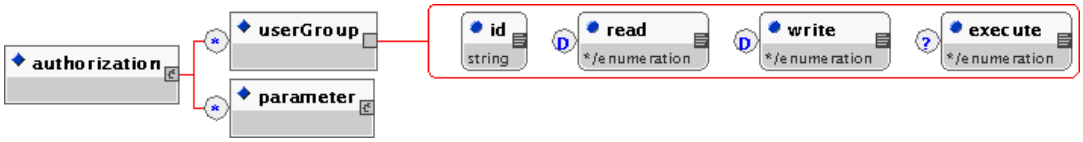

Abbildung 14: Der Elementtyp *authorization* der GResourceDL mit einigen Unterelementen.

`<authorization>`

`(userGroup* , parameter*)`

Dieser Elementtyp stellt eine vorübergehende Lösung zur Einbindung von Autorisierung in die GResourceDL dar (Abbildung 14). Die Ausarbeitung eines Konzeptes für die Autorisierung liegt beim Fraunhofer SIT. Mit Hilfe des Elementtyps `userGroup` lassen sich ähnlich wie bei UNIX-Dateisystemen verschiedene Rechte für verschiedene Benutzergruppen vergeben. Während bei UNIX-Dateisystemen jedoch nur zwischen *owner, group* und *all* unterschieden wird, sind hier frei definierbare Gruppenzugehörigkeiten möglich. Über das Attribut `id` muss eine eindeutige Zuordnung zu einer bestimmten Benutzergruppe möglich sein, deren Mitglieder anderswo zu definieren sind. Zudem gibt es drei weitere Attribute, die für jede Benutzergruppe die Rechte vorgeben:

`read` Ist die Benutzergruppe berechtigt, diese Ressourcenbeschreibung zu lesen und erhält sie Leserecht für die zugehörige Ressource? (Standardwert=`true`)

`write` Darf die Benutzergruppe diese Ressourcenbeschreibung verändern und hat sie Schreibrechte für die zugehörige Ressource? (Standardwert=`false`)

`execute` Darf die Benutzergruppe die Ressource ausführen? (optional)

Hier ein Beispiel für die Angabe von Autorisierungsdaten für verschiedene Benutzergruppen:

```
<resource type="software" id="ein_Programm_fuer_jeden">
```

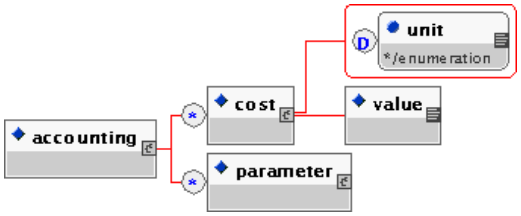

Abbildung 15: Der Elementtyp *accounting* der GResourceDL mit einigen Unterelementen.

```
  ...
  <authorization>
    <userGroup id = "all" read = "true" write = "false" execute = "true"/>
    <userGroup id = "de-fhrg-first" read = "true" write = "true" execute = "true"/>
  </authorization>
</resource>
```

Die Zuordnung einzelner Nutzer zu den hier angegebenen Nutzergruppen-IDs kann zum Beispiel mittels einer Benutzerdatenbank durchgeführt werden.

Das Element `<authorization>` sollte aus Sicherheitsgründen nicht in der GResourceDL enthalten sein, die dem JobBuilder vorliegt. Der JobHandler muss bei einer Autorisierungkomponente nachfragen können, ob Benutzer XY auf die entsprechende Ressource (lesend/schreibend/ausführend) zugreifen darf. Der Nutzer sollte über den JobBuilder oder den Task Mapper generell nur auf Ressourcenbeschreibungen zugreifen können, für die der Nutzer auch eine Leseberechtigung hat. Authentifizierung erfolgt durch Proxy (SIT), die Autorisierung muss beim Submitten erfolgen.

`<accounting>`

`(cost* , parameter*)`

Dieser Elementtyp stellt eine vorübergehende Lösung zur Berücksichtigung eines Abrechnungssystems in der GResourceDL dar (Abbildung 15). Die Ausarbeitung eines Konzeptes für das *accounting* liegt beim Fraunhofer SIT. Mit Hilfe des Elementtyps `cost` können die Gebühren, die bei der Benutzung anfallen, angegeben werden. Für alle Benutzergruppen kann bei diesem Ansatz nur ein einziger Gebührensatz angegeben werden. Die Gebühren sind an den Besitzer der Ressource (siehe `ownerInformation`) auszuzahlen. Damit ein Vergleich der Kosten möglich ist, wurde der eigentliche Wert mittels des Elementtyps `value` gekapselt. Dadurch kann mit Hilfe einer lokalen Ressourcenbeschreibung (siehe `dependencies`) z. B. angegeben werden, dass eine Softwarekomponente nur auf Hardwareressourcen ausgeführt werden soll, deren Verwendung weniger als $5 Euro/h_{CPU}$ kostet. Durch das Attribut `unit` kann angegeben werden, ob pro Verwendung (`EURO_per_use`) oder pro Stunde CPU-Zeit (`EURO_per_cpu_h`, Standardwert) abgerechnet wird. Dieses Konzept muss insbesondere noch dahingehend ausgeweitet werden, dass es möglich sein muss, für verschiedene Benutzergruppen unterschiedliche Benutzungsgebühren erheben zu können.

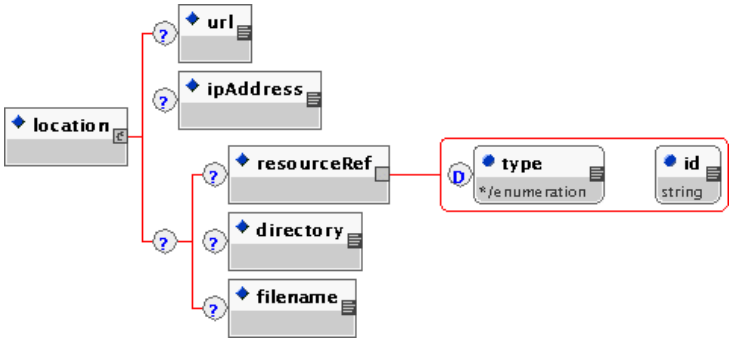

Abbildung 16: Der Elementtyp *location* der GResourceDL mit einigen Unterelementen.

`<location>`

`(url? , ipAddress? , (resourceRef? , directory? , filename?)?)`

Es gibt verschiedene Möglichkeiten, den physischen oder logischen Ort eines Objektes anzugeben (Abbildung 16). Hardwareressourcen können oft durch die Angabe einer IP-Nummer (`ipAddress`) oder einer URL eindeutig lokalisiert werden. Der Zugriff auf Softwarekomponenten oder Daten erfolgt in der Regel über eine URL (`url`) oder über eine Referenz auf die Hardwareressource, auf der die ausführbaren Dateien oder die Daten zu finden sind (`resourceRef`). Dabei kann es notwendig sein, zusätzlich ein oder mehrere Verzeichnisse (`directory`) und/oder Dateinamen (`filename`) anzugeben. Beispiel einer Ortsangabe einer Hardwareressource:

```
<resource id = "HW02_000001_de-fhrg-first_harlekin" type = "hardware">
  ...
  <location>
    <url>x-gram://harlekin.first.gmd.de:2119/jobmanager:
         /O=Grid/O=Globus/CN=harlekin.first.gmd.de</url>
    <ipAddress>194.95.169.11</ipAddress>
  </location>
</resource>
```

`<status>`

`(available?)`

Mit diesem Elementtyp kann der Status der Ressource angegeben werden (Abbildung 17), also ob die Ressource generell verfügbar ist (`<available is="true">`) oder nicht (`<available is="false">`). Weitere Status können hier bei Bedarf in späteren Versionen der GADL hinzugefügt werden.

Abbildung 17: Der Elementtyp *status* der GResourceDL mit dem Unterelement *available*.

Abbildung 18: Der Elementtyp *taskMapping* der GResourceDL mit dem Unterelement *taskId*.

```
<taskMapping>

(taskId*)
```

Der Elementtyp `taskMapping` enthält Informationen, die für die Abbildung einer Problemstellung auf eine Menge von Ressourcen von der Task-Mapping-Engine benötigt werden (Abbildung 18). Die Angabe von Koordinaten möglicher Aufgabengebiete erfolgt über das Unterelement `taskId` in der Form `<taskId>longint, longint</taskId>`. Der erste *longint* steht hierbei für die Aufgabenebene (*task level*), die zweite Zahl steht für die Elementnummer der Aufgabe innerhalb dieser Ebene.

```
<hardwareSpecific>

(benchmark.hardware* , memory.totalPhysical? , memory.freePhysical? ,
 memory.totalVirtual? , memory.freeVirtual? , transmissionSpeed? ,
 filesystem* , cpu.loadAverage* , parameter*)
```

Der Elementtyp `hardwareSpecific` enthält Informationen, die nur für Ressourcen vom Typ `hardware` oder `hardwareClass` relevant sind (Abbildung 19). Folgende optionale Unterelemente können im Elementtyp `hardwareSpecific` enthalten sein:

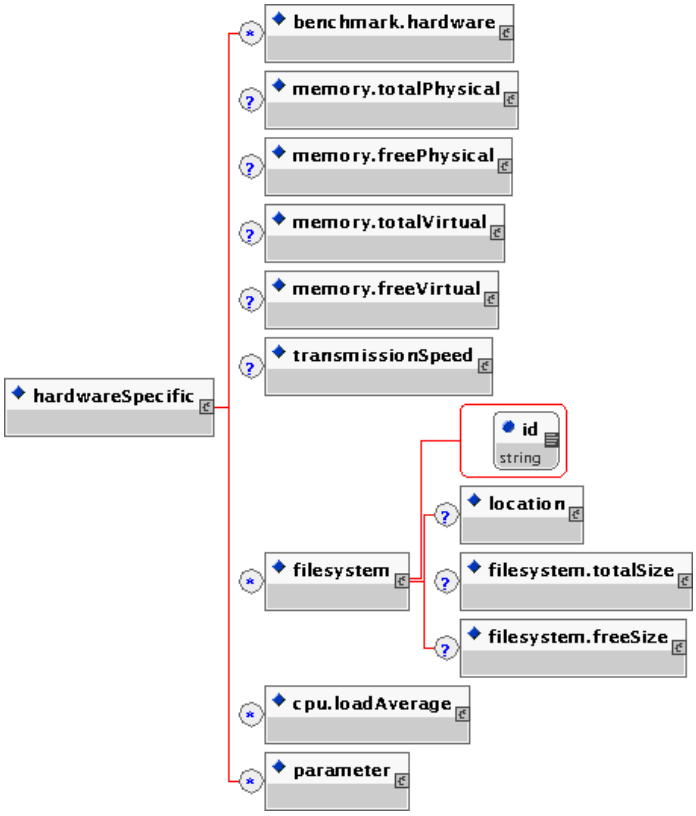

Abbildung 19: Der Elementtyp *hardwareSpecific* der GResourceDL mit Unterelementen.

`benchmark.hardware`	Ergebnis einer Messung der Hardware-Performance bezüglich einer bestimmten Metrik (→ ITWM?) z. B. SPEC oder Lapack [*von Laszewski*, 2002]. Die Einheit sollte so gewählt werden, dass $\frac{benchmark.software}{benchmark.hardware}$ die voraussichtliche Ausführungsdauer in s ergibt
`memory.totalPhysical`	Gesamter Speicherplatz des RAM-Speichers
`memory.freePhysical`	Freier Speicherplatz des RAM-Speichers
`memory.totalVirtual`	Gesamter virtueller Speicherplatz
`memory.freeVirtual`	Freier virtueller Speicherplatz
`transmissionSpeed`	Übertragungsgeschwindigkeit mit bzw. durch die Hardwareressource. Oftmals auch (fälschlicherweise) Bandbreite genannt (eigentlich Frequenzbereich in Hz). Standardeinheit: bit/s (DIN 44302)
`filesystem`	Informationen über gemountete Dateisysteme
`cpu.loadAverage`	Mittlere CPU-Auslastung in der vergangenen Zeit
`parameter`	Beliebige weitere Parameter

Mit dem Elementtyp `benchmark.hardware` muss ein Attribut `type` angegeben werden. Mögliche Werte für `type` sind `spec`, `lapack.oneHundred`, `lapack.fiveHundred` und `lapack.thousand`. Es können mehrere Dateisysteme pro Hardwareressource mit Hilfe von `filesystem` beschrieben werden. Das Attribut `id` muss eine eindeutige Zuordnung des

Dateisystems innerhalb der Ressource ermöglichen. Mittels `location` wird der *Mountpoint* des Dateisystems angegeben. `filesystem.totalSize` gibt den gesamten, `filesystem.freeSize` den freien Speicherplatz an. Die Angabe der mittleren CPU-Auslastung (`cpu.loadAverage`) erfolgt je nach Wert des Attributs `type` bezüglich der letzten Minute (`pastOneMinute`), bezüglich der letzten fünf Minuten (`pastFiveMinutes`) oder bezüglich der letzten zehn Minuten (`pastTenMinutes`). Hier ein Beispiel:

```
<resource id = "HW02_000001_de-fhrg-first_harlekin" type = "hardware">
  ...
  <hardwareSpecific>
    <memory.totalPhysical unit = "MB">
      <value type = "int">1005</value>
    </memory.totalPhysical>
    <memory.freePhysical unit = "MB">
      <value type = "int">793</value>
    </memory.freePhysical>
    <memory.totalVirtual unit = "MB">
      <value type = "int">1027</value>
    </memory.totalVirtual>
    <memory.freeVirtual unit = "MB">
      <value type = "int">1027</value>
    </memory.freeVirtual>
    <filesystem id = "/usr">
      <location>
        <directory>/usr</directory>
      </location>
      <filesystem.totalSize unit = "MB">
        <value type = "int">9844</value>
      </filesystem.totalSize>
      <filesystem.freeSize unit = "MB">
        <value type = "int">7836</value>
      </filesystem.freeSize>
    </filesystem>
  </hardwareSpecific>
</resource>
```

`<softwareSpecific.static>`

`(input* , output* , arguments? , gidl? , executable? , sourceCode? , developmentTool? , realtimeBehaviour? , benchmark.software? , parameter*)`

Der Elementtyp `softwareSpecific.static` enthält statische Informationen über Ressourcen vom Typ `software` oder `softwareClass` (Abbildung 20). Unter *statischen* Informationen sind diejenigen Parameter zu verstehen, die in allen Instanzen einer GResourceDL-Beschreibung der selben Ressource identisch sind, sich also in verschiedenen XML-Dokumenten nicht unterscheiden. Diese Parameter dürfen somit z. B. nicht durch den Benutzer abgeändert werden, nachdem die Ressourcenbeschreibung aus dem Repository geholt wurde. Statische Parameter können in diesem Sinne durchaus zeitlich variabel

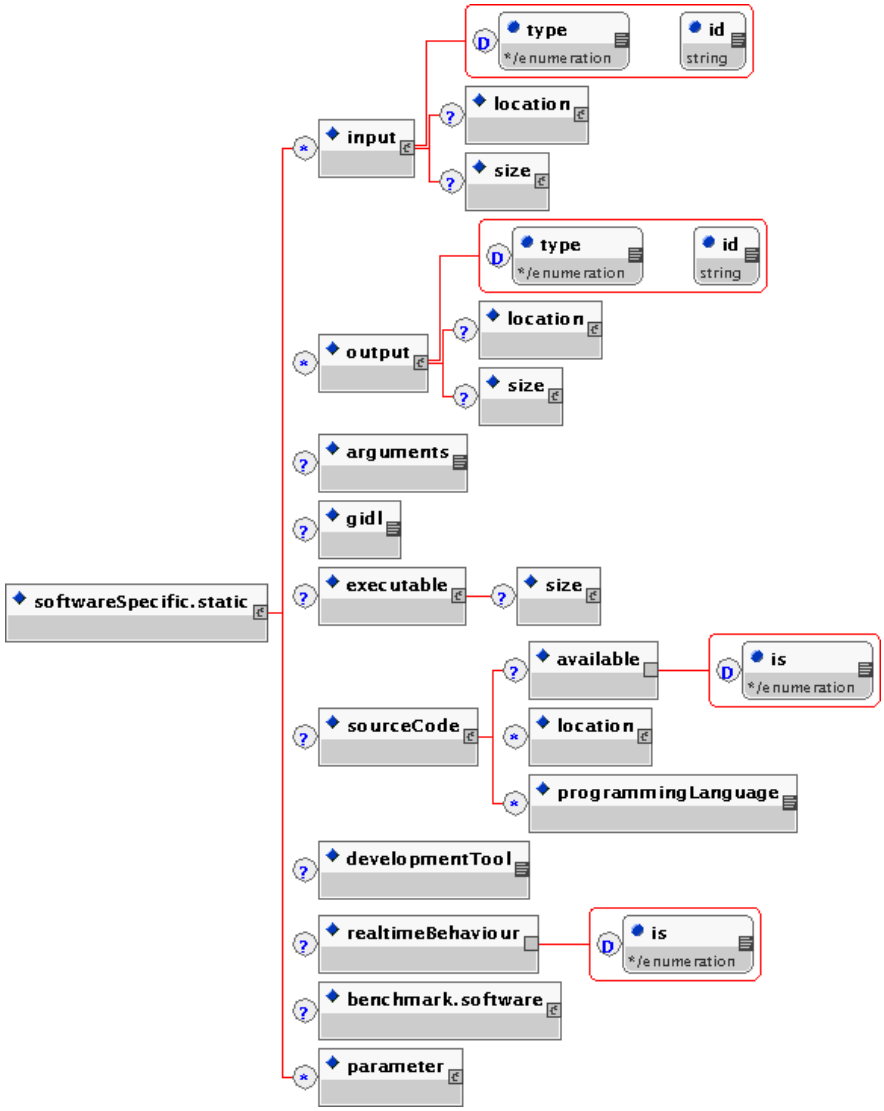

Abbildung 20: Der Elementtyp *softwareSpecific.static* der GResourceDL mit Unterelementen.

sein (analog zum Variablentyp *static* bei Java), die Änderung muss sich jedoch auf alle Instanzen beziehen. Variablen, die vom Benutzer oder der Grid-Architektur während der Abarbeitung einer Grid-Anwendung zu verändern sind, werden mit dem Elementtyp `softwareSpecific.variable` beschrieben.

Folgende optionale Unterelemente können im Elementtyp `softwareSpecific.static` enthalten sein:

`input`	Fester Eingabe-Port der Softwarekomponente (stdin, Datei)
`output`	Fester Ausgabe-Port der Softwarekomponente (stdout, stderr, Datei)
`arguments`	Feste Kommandozeilenparameter
`gidl`	Verweis auf eine Schnittstellenbeschreibung im GInterfaceDL-Format (wird noch nicht unterstützt)
`executable`	Informationen über die ausführbare Datei (z. B. Dateigröße). Der Ort an dem die ausführbare Datei zu finden ist wird nicht hier, sondern unter `location` (siehe Abbildung 8) abgelegt
`sourceCode`	Informationen über den Quellcode
`developmentTool`	Das Werkzeug, das verwendet wurde, um die Softwarekomponente zu entwickeln
`realtimeBehaviour`	Echtzeitverhalten der Softwarekomponente, z. B. wenn zur Laufzeit ein Video-Stream erzeugt wird.
`benchmark.software`	Ergebnisse einer CPU-Zeitmessung auf einer definierten Hardwareumgebung ($\rightarrow$ ITWM?). Diese Daten werden für das Scheduling benötigt. Die Einheit sollte so gewählt werden, dass $\frac{benchmark.software}{benchmark.hardware}$ die voraussichtliche Ausführungsdauer in s ergibt
`parameter`	Beliebige weitere Parameter

Die für die Ausführung von gekoppelten Grid-Anwendungen wichtigsten Elemente sind die Angaben über die Eingabe- (`input`) und Ausgabe-Ports (`output`) der Softwarekomponente. Die Art des Ports wird mittels des Attributs `type` vorgegeben. Als Eingabe-Ports werden zur Zeit Standard-Eingabe (`stdin`) und Eingabe-Datei (`file`) unterstützt. Der Ort der Eingabe-Datei (`location`) kann entweder als Unterelement von `softwareSpecific.static` — d. h. der Ort und Name der Eingabedatei ist fest vorgegeben — oder als Unterelement von `softwareSpecific.variable` vorgegeben werden, was bedeutet, dass die Softwarekomponente es erlaubt, dass der Benutzer den Ort und Namen der Eingabe-Datei frei vorgibt. Es ist dann Aufgabe der Grid-Architektur, die benötigten Eingabe-Daten an die angegebene Stelle zu kopieren. Das Attribut `id` muss eine eindeutige Identifizierung des Ports innerhalb der Ressource ermöglichen.

Analoges gilt für die Beschreibung der Ausgabe-Ports der Softwarekomponente (`output`). Zur Zeit werden Standard-Ausgabe (`stdout`), Standard-Error (`stderr`) und Ausgabe-Dateien (`file`) unterstützt.

`<softwareSpecific.variable>`

`(executionLocation? , input* , output* , arguments? , parameter*)`

Dieser Elementtyp enthält softwarespezifische Variablen, die bei jeder Ausführung der selben Softwarekomponente unterschiedliche Werte annehmen können (Abbildung 21). Die Werte werden entweder vom Benutzer beim Zusammenbau der Grid-Anwendung (z. B. mit

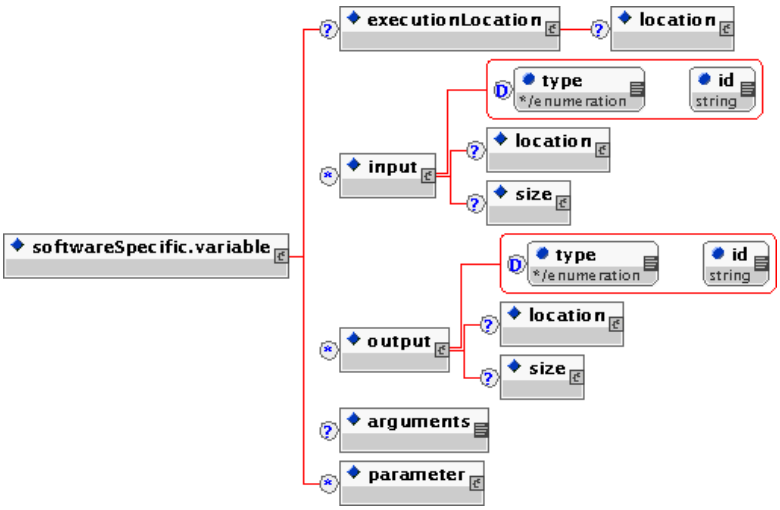

Abbildung 21: Der Elementtyp *softwareSpecific.variable* der GResourceDL mit Unterelementen.

Hilfe des JobBuilders) oder von anderen Komponenten der Grid-Architektur (z. B. Meta-Scheduler) gesetzt. Folgende Unterelemente sind möglich:

`executionLocation`	Ort der Hardwareressource, auf der die Softwarekomponente ausgeführt werden soll (üblicherweise Angabe einer URL in Form von `x-gram://.../jobmanager:/...`). Dieser Eintrag wird vom JobHandler verwendet, um die Softwarekomponente zu submitten.
`input`	Variabler Eingabe-Port der Softwarekomponente (Datei), der zur Laufzeit z. B. per Kommandozeilenparameter festgelegt werden kann
`output`	Variabler Ausgabe-Port der Softwarekomponente (Datei), der zur Laufzeit z. B. per Kommandozeilenparameter festgelegt werden kann
`arguments`	Zusätzliche Kommandozeilenparameter, die vom Benutzer vorgegeben werden
`parameter`	Beliebige weitere Parameter

Werden mittels `input` oder `output` variable Eingabe- bzw. Ausgabedateien der Softwarekomponente festgelegt, so muss es ein für alle Softwarekomponenten einheitliches Protokoll geben, mit dem der Softwarekomponente die Dateinamen und Verzeichnisse der Ein- und Ausgabedateien mitgeteilt werden können. Denkbar wäre zum Beispiel, dass man die entsprechende Softwarekomponente mit einem einfachen Shellskript kapselt, das einheitliche Kommandozeilenparameter in die jeweils softwarespezifische Syntax umwandelt. Hier ein Vorschlag für einheitliche Kommandozeilenparameter zur Angabe der Ein- und Ausgabedateien:

```
-input <inputId> <filename> [<directory>]
-output <outputId> <filename> [<directory>]
```

Im Folgenden sollen die verschiedenen Schritte anhand eines Beispiels erläutert werden. Die im Repository abgelegte Ressourcenbeschreibung einer Softwarekomponente, die z. B. aus zwei Eingabedateien mit Informationen über den Niederschlag (precipitation) und den Boden (soil) eine Ausgabedatei mit der Biomasse von Pflanzen erstellt, sieht dann so aus:

```
<fhrgResources>
  <resource id = "SW02_000099_de-fhrg-first_biomassSim" type = "software">
    ...
    <location>
      <resourceRef id = "HW02_000001_de-fhrg-first_harlekin" type = "hardware"/>
      <directory>/home/fhrg/bin</directory>
      <filename>biomassSim.sh</filename>
    </location>
    ...
    <softwareSpecific.static>
      <output id = "stdout" type = "stdout"/>
      <output id = "stderr" type = "stderr"/>
      ...
    </softwareSpecific.static>
    <softwareSpecific.variable>
      <input id = "precipitation" type = "file"/>
      <input id = "soil" type = "file"/>
      <output id = "biomass" type = "file"/>
    </softwareSpecific.variable>
  </resource>
</fhrgResources>
```

Die möglichen Eingabe- und Ausgabedateien sind zwar schon angegeben, die konkreten Dateinamen werden aber erst durch den Benutzer bzw. die Grid-Architektur definiert. Der Ort der Hardwareressource, auf dem diese Softwarekomponente ausgeführt werden soll, wird anschließend z. B. durch einen Meta-Scheduler ausgewählt. Für diese konkrete Ausführung der Softwarekomponente wird die Ressourcenbeschreibung somit um einige Unterelemente des Elementtyps `softwareSpecific.variable` ergänzt:

```
<fhrgResources>
  <resource id = "SW02_000099_de-fhrg-first_biomassSim" type = "software">
    ...
    <softwareSpecific.variable>
      <executionLocation>
        <location>
          <url>x-gram://harlekin.first.gmd.de:2119/jobmanager:
               /O=Grid/O=Globus/CN=harlekin.first.gmd.de</url>
        </location>
      </executionLocation>
      <input id = "precipitation" type = "file">
        <location>
          <directory>data</directory>
          <filename>pr.dat</filename>
        </location>
      </input>
      <input id = "soil" type = "file">
        <location>
          <directory>data</directory>
          <filename>so.dat</filename>
        </location>
```

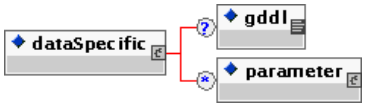

Abbildung 22: Der Elementtyp *dataSpecific* der GResourceDL mit Unterelementen.

```
      </input>
      <output id = "biomass" type = "file">
        <location>
          <directory>output</directory>
          <filename>out.dat</filename>
        </location>
      </output>
    </softwareSpecific.variable>
  </resource>
</fhrgResources>
```

Damit sind nun alle Informationen vorhanden, die der JobHandler benötigt, um die Softwarekomponente z. B. mittels GRAM auf dem Grid auszuführen. Der Aufruf von `biomassSim.sh` mit den Eingabedateien `data/pr.dat` für „precipitation“ und `data/so.dat` für „soil“ und der Ausgabedatei `output/out.dat` mit der berechneten Biomasse könnte dann gemäß obigen Protokolls so aussehen:

```
> biomassSim.sh -input precipitation pr.dat data \
                -input soil so.dat data \
                -output biomass out.dat output
```

`<dataSpecific>`

`(gddl? , parameter*)`

In diesem Elementtyp werden Informationen abgelegt, die spezifisch für Ressourcen vom Typ `data` oder `dataClass` sind (Abbildung 22). Zur Zeit sind die einzigen möglichen Unterelemente `gddl` und `parameter`. `gddl` ist ein Verweis auf die Beschreibung der Daten im GDataDL-Format. Mit Hilfe von `parameter` können beliebige weitere Parameter angegeben werden.

3.1.3 Beispiele

Beispiel einer XML-Ressourcenbeschreibung der Hardwareklasse „x86“:

```
<?xml version = "1.0" encoding = "ISO-8859-1"?>
<!DOCTYPE fhrgResources SYSTEM "grdl0_2.dtd">
<fhrgResources>
  <resource id = "HC00_000001_de-fhrg_x86" type = "hardwareClass">
    <identification>
      <description>
```

```
        <short>x86</short>
        <detailed>Hardware with x86-compatible processor</detailed>
      </description>
    </identification>
    <authorization>
      <userGroup id = "all" read = "true" write = "false"/>
    </authorization>
  </resource>
</fhrgResources>
```

Beispiel einer GResourceDL-Beschreibung der Hardwareklasse „Ethernet 100MBit/s Netzwerkkarte“:

```
<?xml version = "1.0" encoding = "ISO-8859-1"?>
<!DOCTYPE fhrgResources SYSTEM "grdl0_2.dtd">
<fhrgResources>
  <resource id = "HC00_000002_de-fhrg_network-ethernet-100" type = "hardwareClass">
    <identification>
      <description>
        <short>Ethernet network card with 100MBit/s</short>
        <detailed>Ethernet network interface card with transmission speed of 100MBit/s</detailed>
      </description>
      <keyword>ethernet</keyword>
      <keyword>network interface card</keyword>
    </identification>
    <authorization>
      <userGroup id = "all" read = "true" write = "false"/>
    </authorization>
    <hardwareSpecific>
      <transmissionSpeed unit = "MBit_per_s">
        <value type = "int">100</value>
      </transmissionSpeed>
    </hardwareSpecific>
  </resource>
</fhrgResources>
```

Beispiel einer GResourceDL-Beschreibung des Rechners „harlekin.first.gmd.de“. Diese Beschreibung der Hardware erbt durch Erweiterung die Eigenschaften verschiedener Hardwareklassen (x86, network-ethernet-100) und stellt verschiedene Klassen von Software zur Verfügung (linux-kernel-2-4-18, glibc6):

```
<?xml version = "1.0" encoding = "ISO-8859-1"?>
<!DOCTYPE fhrgResources SYSTEM "grdl0_2.dtd">
<fhrgResources>
  <resource id = "HW02_000001_de-fhrg-first_harlekin" type = "hardware">
    <identification>
      <description>
        <short>harlekin.first.gmd.de</short>
        <detailed>Linux PC harlekin at Fraunhofer FIRST</detailed>
      </description>
      <keyword>globus node</keyword>
      <keyword>PC</keyword>
    </identification>
    <dependencies type = "provides">
```

```
      <resourceRef id = "HC00_000001_de-fhrg_x86" type = "hardwareClass"/>
      <resourceRef id = "HC00_000002_de-fhrg_network-ethernet-100" type = "hardwareClass"/>
      <resourceRef id = "SC00_000002_de-fhrg_linux-kernel-2-4-18" type = "softwareClass"/>
      <resourceRef id = "SC00_000003_de-fhrg_glibc6" type = "softwareClass"/>
    </dependencies>
    <authorization>
      <userGroup id = "all" read = "true" write = "false"/>
      <userGroup id = "de-fhrg-first" read = "true" write = "true" execute = "true"/>
    </authorization>
    <accounting>
      <cost unit = "EURO_per_cpu_h">
        <value type = "double" op = "eq">0</value>
      </cost>
    </accounting>
    <location>
      <url>x-gram://harlekin.first.gmd.de:2119/jobmanager:
           /O=Grid/O=Globus/CN=harlekin.first.gmd.de</url>
      <ipAddress>194.95.169.11</ipAddress>
    </location>
    <status>
      <available is = "true"/>
    </status>
    <hardwareSpecific>
      <memory.totalPhysical unit = "MB">
        <value type = "int">1005</value>
      </memory.totalPhysical>
      <memory.freePhysical unit = "MB">
        <value type = "int">793</value>
      </memory.freePhysical>
      <memory.totalVirtual unit = "MB">
        <value type = "int">1027</value>
      </memory.totalVirtual>
      <memory.freeVirtual unit = "MB">
        <value type = "int">1027</value>
      </memory.freeVirtual>
      <filesystem id = "/usr">
        <location>
          <directory>/usr</directory>
        </location>
        <filesystem.totalSize unit = "MB">
          <value type = "int">9844</value>
        </filesystem.totalSize>
        <filesystem.freeSize unit = "MB">
          <value type = "int">7836</value>
        </filesystem.freeSize>
      </filesystem>
    </hardwareSpecific>
  </resource>
</fhrgResources>
```

GResourceDL-Beschreibung der Softwareklasse „Linux“:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE fhrgResources SYSTEM "grdl0_2.dtd">
<fhrgResources>
  <resource type = "softwareClass" id = "SC000100001_de-fhrg_linux">
```

```
    <identification>
      <description>
        <short>linux operating system</short>
      </description>
      <keyword>linux</keyword>
      <keyword>operating system</keyword>
    </identification>
    <authorization>
      <userGroup id = "all" read = "true" write = "false"/>
    </authorization>
  </resource>
</fhrgResources>
```

Die GResourceDL-Beschreibung der Softwareklasse „linux-kernel-2-4-18“ erbt die Eigenschaften der Softwareklasse „linux“:

```
<?xml version = "1.0" encoding = "ISO-8859-1"?>
<!DOCTYPE fhrgResources SYSTEM "grdl0_2.dtd">
<fhrgResources>
  <resource id = "SC00_000002_de-fhrg_linux-kernel-2-4-18" type = "softwareClass">
    <identification>
      <description>
        <short>linux with kernel version 2.4.18</short>
      </description>
    </identification>
    <dependencies type = "provides">
      <resourceRef id = "SC00_000001_de-fhrg_linux" type = "softwareClass"/>
    </dependencies>
    <authorization>
      <userGroup id = "all" read = "true" write = "false"/>
    </authorization>
  </resource>
</fhrgResources>
```

Die GResourceDL-Beschreibung der Softwareklasse „glibc6“:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE fhrgResources SYSTEM "grdl0_2.dtd">
<fhrgResources>
  <resource type = "softwareClass" id = "SC00_000003_de-fhrg_glibc6">
    <identification>
      <description>
        <short>glibc6</short>
      </description>
      <keyword>libc</keyword>
      <keyword>GNU</keyword>
    </identification>
    <manufacturerInformation>
      <name>Sleepycat Software</name>
      <contact>
        <name>Sleepycat Software</name>
        <email>db@sleepycat.com</email>
      </contact>
      <version>6</version>
    </manufacturerInformation>
```

```
    <authorization>
      <userGroup id = "all" read = "true" write = "false"/>
    </authorization>
  </resource>
</fhrgResources>
```

Die GResourceDL-Beschreibung der konkreten Softwarekomponente „sleep“, die sich auf dem Rechner „harlekin“ befindet und einen Speicherbedarf von 1912 Byte hat:

```
<?xml version = "1.0" encoding = "ISO-8859-1"?>
<!DOCTYPE fhrgResources SYSTEM "grdl0_2.dtd">
<fhrgResources>
  <resource id = "SW02_000001_de-fhrg-first_sleep" type = "software">
    <identification>
      <description>
        <short>harlekin.first.gmd.de:/bin/sleep</short>
        <detailed>sleep - delay for a specified amount of time</detailed>
      </description>
      <keyword>linux</keyword>
      <keyword>sleep</keyword>
      <keyword>GNU</keyword>
      <keyword>sh-utils</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"/>
      <resource id = "memory.freeVirtual.gt.1912Byte" type = "hardwareClass">
        <hardwareSpecific>
          <memory.freeVirtual unit = "Byte">
            <value op = "gt">1912</value>
          </memory.freeVirtual>
        </hardwareSpecific>
      </resource>
    </dependencies>
    <manufacturerInformation>
      <name>Free Software Foundation, Inc., Jim Meyering and Paul Eggert</name>
      <contact>
        <name>report Bugs to</name>
        <email>bug-sh-utils@gnu.org</email>
      </contact>
      <version>2.0.11</version>
      <parameter name = "GNU package">
        <value type = "string">sh-utils</value>
      </parameter>
    </manufacturerInformation>
    <documentation>
      <documentationRef>
        <location>
          <resourceRef id = "HW02_000001_de-fhrg-first_harlekin" type = "hardware"/>
          <directory>/usr/share/doc/sh-utils-2.0.11</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">0.00</value>
      </cost>
    </accounting>
    <location>
      <resourceRef id = "HW02_000001_de-fhrg-first_harlekin" type = "hardware"/>
      <directory>/bin</directory>
      <filename>sleep</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">11612</value>
        </size>
      </executable>
      <sourceCode>
        <available is = "true"/>
        <programmingLanguage>C</programmingLanguage>
      </sourceCode>
    </softwareSpecific.static>
  </resource>
</fhrgResources>
```

3.2 Grid Interface Definition Language (GInterfaceDL)

Ziel der GInterfaceDL ist es, von den verschiedenen Arten des Online-Zugriffs auf Softwarekomponenten zu abstrahieren und eine allgemeingültige Beschreibung der Schnittstellen zu erhalten, die im Wesentlichen unabhängig von der verwendeten Komponentenumgebung ist. Bei der Ausführung von gekoppelten Grid-Anwendungen muss diese Schnittstellenbeschreibung in die entsprechende Beschreibungssprache (z. B. CORBA-IDL [*Object Management Group*, 2001]) übersetzt werden. Hier ein Beispiel einer CORBA-IDL einer Softwarekomponente mit den von außen zugänglichen Methoden `solve` und `exit`:

```
module FHRG {

typedef float Array2[100][100];
typedef float Array1[100];

interface MatrixSolver {
  boolean solve (in Array2 inputArray, out Array1 outputArray);
  void exit ();
};

};
```

Die GInterfaceDL in dieser Version der GADL wurde noch nicht getestet und es fehlen noch einige wesentlichen Elemente, wie zum Beispiel Fehlerbehandlung (CORBA-IDL:

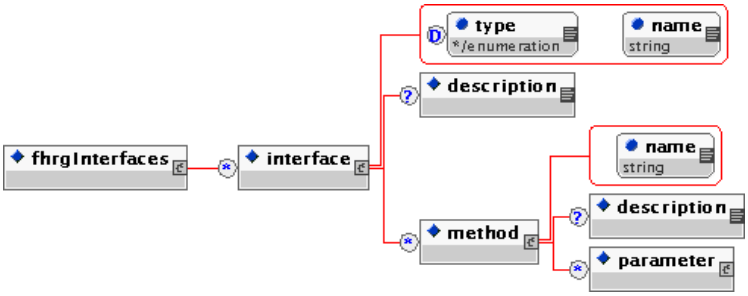

Abbildung 23: Der Elementtyp *fhrgInterfaces* der GInterfaceDL mit einigen Unterelementen.

exception, raises) und zusammengesetzte Typen (CORBA-IDL: struct). Hier ist es erst sinnvoll, konkreter zu werden, wenn die ersten Spezifikationen bezüglich Komponentenarchitektur vorliegen. Mittels der Komponentenarchitektur und der GInterfaceDL soll es möglich werden, auf generische Art und Weise auf die Methoden einer Softwarekomponente zuzugreifen.

3.2.1 Datenmodell

Eine Online-Dokumentation der GInterfaceDL ist unter
http://www.fhrg.fhg.de/de/fhrg/GADL_0_2/docs/gidl0_2_dtd/
zu finden.

```
<fhrgInterfaces>

(interface*)
```

Dieses Element ist das Dokument-Element der GInterfaceDL und muss damit genau einmal in dem Dokument vorkommen (Abbildung 23). Der Inhalt von Element `<fhrgInterfaces>` kann aus null oder mehreren Elementen vom Typ `<interface>` bestehen.

```
<interface type = "..." name = "...">

(description? , method*)
```

Dieser Elementtyp kann als Unterelemente eine Beschreibung (`description`) und die Definition mehrerer Methoden (`method`) enthalten (Abbildung 23). Mittels des Attributs `type` kann angegeben werden, ob es sich um eine CORBA-Schnittstelle (`type = "corba"`) oder um eine SOAP-Schnittstelle (`type = "soap"`) handelt.

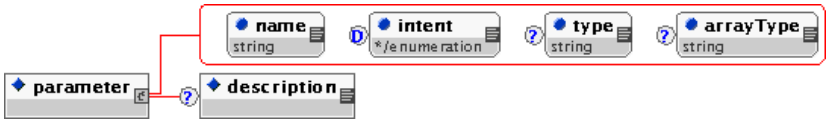

Abbildung 24: Der Elementtyp *parameter* der GInterfaceDL.

```
<method name = "...">

 (description? , parameter*)
```

Dieser Elementtyp kann als Unterelemente eine Beschreibung (`description`) und die Definition mehrerer Eingabe-, Ausgabe- sowie Rückgabeparameter (`parameter`) enthalten (Abbildung 23). Mittels des Attributs `name` wird der Name der Methode angegeben.

```
<parameter name = "..." intent = "..." type = "..." arrayType = "...">

(description?)
```

Mit Hilfe dieses Elementtyps werden die Typen der Eingabe-, Ausgabe- sowie Rückgabewerte der Methoden definiert (Abbildung 24). Das Element `parameter` besitzt folgende Attribute:

`name`	Name des Parameters
`intent`	Ist dieser Parameter Eingabe-, Ausgabe oder Rückgabewert der Methode? Mögliche Werte: `in`=Eingabewert, `out`=Ausgabewert, `inout`=sowohl Eingabe- als auch Ausgabewert, `return`=Rückgabewert
`type`	Parametertyp analog zum Attribut `type` des Elements `value` der GResourceDL: float, double, int, boolean oder string
`arrayType`	Parametertyp und Dimension ein oder mehrdimensionaler Felder, z. B. `float[100,100]` für ein zweidimensionales 100 × 100-Feld vom Typ float

3.2.2 Beispiele

Hier ein Beispiel einer GInterfaceDL-Beschreibung, die auf obige CORBA-IDL abgebildet werden könnte:

```
<?xml version = "1.0" encoding = "ISO-8859-1"?>
<!DOCTYPE fhrgInterfaces SYSTEM "gidl0_2.dtd">
<fhrgInterfaces>
  <interface type = "corba" name = "matrixSolver">
    <description>this is the interface definition for matrixSolver</description>
    <!-- boolean solve( float[100,100] inputArray, float[100] outputArray ) -->
```

```
    <method name = "solve">
      <description>this method solves a linear equation system</description>
      <parameter name = "inputArray" intent = "in" arrayType = "float[100,100]"/>
      <parameter name = "outputArray" intent = "out" arrayType = "float[100]"/>
      <parameter name = "returnValue" intent = "return" type = "boolean"/>
    </method>
    <!-- void exit() -->
    <method name = "exit">
      <description>this method exits matrixSolver</description>
    </method>
  </interface>
</fhrgInterfaces>
```

3.3 Grid Data Definition Language (GDataDL)

Die GDataDL wird von der Grid-Architektur des Fraunhofer Resource Grids zur Zeit noch nicht unterstützt. Hier sollen nur grundsätzliche Ideen zur Konzeption einer Beschreibungssprache für Dateninhalte und Datenformate festgehalten werden.

Die GDataDL kann mehreren Zwecken dienen. Zum einen kann sie dazu verwendet werden, den Inhalt und das Datenformat von Dateien zu beschreiben, die Daten in binärer oder in textkodierter Form enthalten. Diese Beschreibung kann dann z. B. dazu verwendet werden zu testen, ob Eingabedateien kompatibel zu der Softwarekomponente sind, die mit den Eingabedaten verknüpft wurde. Zudem können anhand dieser Beschreibung Filter generiert werden, die eine automatische Konvertierung verschiedener Formate ermöglichen.

Eine andere Möglichkeit zur Vereinfachung des Datenaustausches zwischen den jeweiligen Softwarekomponenten wird dadurch erreicht, dass die Daten selbst im GDataDL-Format (selbstbeschreibende Daten) von den Programmen ausgegeben werden, sodass jede Softwarekomponente mit Hilfe eines XML-Parsers die Daten einer anderen Komponente einlesen und „verstehen“ kann. Die GDataDL beschreibt dabei nicht nur das Format (float, double, integer, ...), sondern auch z. B. die physikalische Bedeutung (Wärmefluss durch Fläche XY in Einheiten J/m^2) der Daten.

Eine andere Möglichkeit ist die Trennung von Format und Daten. Während das Format der Daten mittels XML beschrieben wird, werden die Daten selber unverändert (z. B. binär) übertragen.

Daten im GDataDL-Format können auch Methodenaufrufe oder Fehlermeldungen (ähnlich wie bei SOAP) enthalten. Softwarekomponenten, welche GDataDL ausgeben und verstehen, können sehr einfach online gekoppelt werden, indem der Ausgabe-Stream (z. B. stdout) des einen Programms mit dem Eingabe-Stream (z. B. stdin) des anderen Programms verbunden wird. Dies kann durch eine einfache Socket-Verbindung erfolgen.

3.3.1 Datenmodell

Eine Online-Dokumentation der GDataDL ist unter
http://www.fhrg.fhg.de/de/fhrg/GADL_0_2/docs/gddl0_2_dtd/
zu finden.

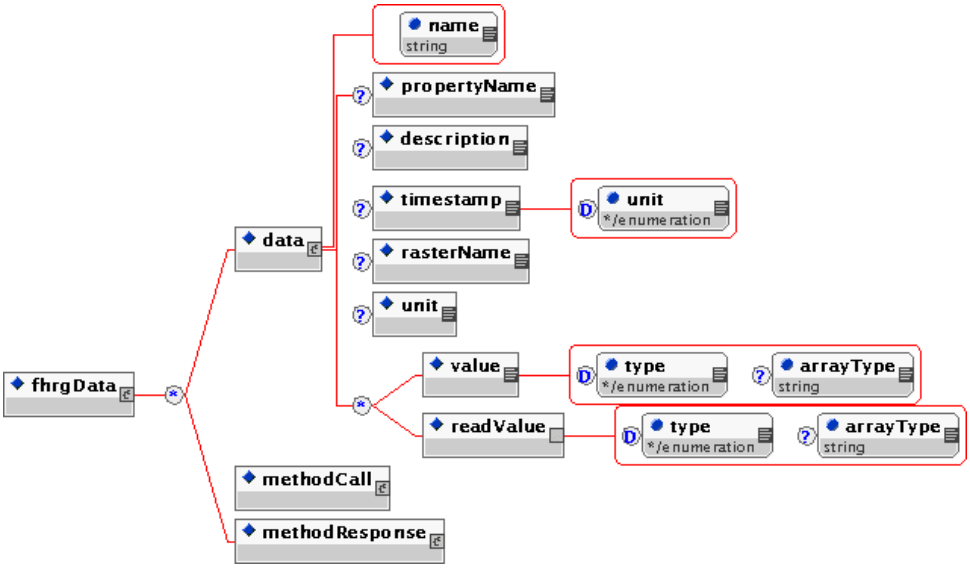

Abbildung 25: Der Elementtyp *fhrgData* der GDataDL.

`<fhrgData>`

`(data | methodCall | methodResponse)*`

Dieses Element ist das Dokument-Element der GDataDL und muss damit genau einmal in dem Dokument vorkommen (Abbildung 25). Der Inhalt von Element `<fhrgData>` kann aus null oder mehreren Elementen vom Typ `<data>`, `<methodCall>` oder `<methodResponse>` bestehen.

`<data>`

```
(propertyName? , description? , timestamp? , rasterName? , unit? ,
 (value | readValue)*)
```

Mit diesem Elementtyp werden das Format und der Inhalt einzelner Daten beschrieben (Abbildung 25). `propertyName` enthält die (z. B. physikalische) Bezeichnung des Parameters. Mit `description` kann eine kurze Beschreibung angegeben werden. Mit der Angabe der Zeit `timestamp` kann analog zum Elementtyp `timestamp` der GResourceDL (siehe Seite 19) der Zeitpunkt angegeben werden, dem die Werte des Parameters zugeordnet sind. Mit Hilfe des Elementtyps `rasterName` kann gegebenenfalls angeben werden, bezüglich welchen Rasters die Daten angegeben sind. Mit `unit` ist die (physikalische) Einheit der Werte gemeint. Als Standard sollten hier möglichst SI-Einheiten verwendet werden, um Kompatibilität zwischen verschiedenen Softwarekomponenten zu ermöglichen. Falls die Werte selber Bestandteil der GDataDL-Beschreibung sind, so werden diese mittels des Elementtyps `value` angegeben; ansonsten wird mit `readValue` das Format festgelegt, in dem die Werte an anderer Stelle gespeichert sind.

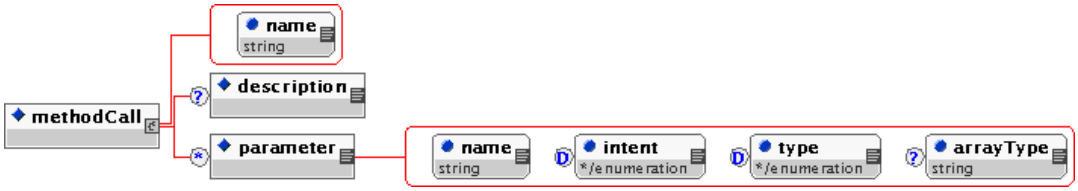

Abbildung 26: Der Elementtyp *methodCall* der GDataDL.

```
<value type = "..." arrayType = "..." >

(#PCDATA)
```

Mittels des Attributs `type` kann der Typ eines skalaren Wertes angegeben werden. Mögliche Typen sind analog zur GResourceDL (siehe Seite 19) float, double, int, boolean, string und base64. Das Attribut `arrayType` kann verwendet werden, um Felder zu definieren. Die Syntax ist dabei ähnlich zu der in SOAP [*Box et al.*, 2000]. Mit `float[10,20]` kann z. B. ein zweidimensionales 10×20-Feld von Fließkommazahlen mit einfacher Genauigkeit definiert werden.

```
<readValue type = "..." arrayType = "..." >

EMPTY
```

Der Elementtyp `readValue` ist ein Leeres-Element-Tag, das entweder mit dem Attribut `type` oder `arrayType` versehen wird. Werte, deren Format mit `readValue` beschrieben werden, sind nicht in dem GDataDL-Dokument selber enthalten, sondern sind an anderer Stelle gespeichert. Der Ort, von dem die Daten gelesen werden können, ist in der zugehörigen GResourceDL-Beschreibung der Daten durch das Element `<location>` festgelegt.

```
<methodCall name = "...">

(description? , parameter*)
```

Mit diesem Elementtyp können Methodenaufrufe als Bestandteil von Eingabedaten von Softwarekomponenten definiert werden (Abbildung 26). Für das Auswerten der in GDataDL kodierten Methodenaufrufe ist die Softwarekomponente selber zuständig, das heißt, sie muss mit einem XML-Parser die Eingabedaten gemäß der GDataDL-Syntax auswerten und die entsprechenden lokalen Methoden aufrufen. Mit Hilfe von `methodCall` können dann Softwarekomponenten von außen mit einer Art Skriptsprache gesteuert werden, ohne dass eine spezielle Komponentenumgebung, wie z. B. CORBA oder SOAP vorhanden sein muss. Das Unterelement `parameter` ist analog zur GInterfaceDL (siehe Seite 41) zu verwenden.

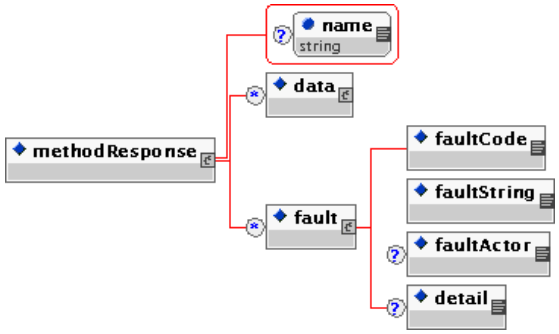

Abbildung 27: Der Elementtyp *methodResponse* der GDataDL.

```
<methodResponse name = "...">

(data* , fault*)
```

Mit diesem Elementtyp werden die Rückgabewerte von Methodenaufrufen als Bestandteil der Ausgabedaten einer Softwarekomponente gekapselt (Abbildung 27). Für das Erzeugen dieser Rückgabewerte ist die Softwarekomponente selber verantwortlich. Neben den Ausgabedaten können auch Fehlermeldungen mit Hilfe des Elementtyps `fault` analog zur SOAP-Syntax [*Box et al.*, 2000] übermittelt werden.

3.3.2 Beispiele

Hier ein Beispiel einer GDataDL-Beschreibung, die mehrere Unterelemente vom Typ `data`, `methodCall` und `methodResponse` enthält.

```
<?xml version = "1.0" encoding = "ISO-8859-1"?>
<!DOCTYPE fhrgData SYSTEM "gddl0_2.dtd">
<fhrgData>
  <!-- base64 encoded data -->
  <data name = "picture">
    <description>this is binary data with base64 encoding</description>
    <timestamp unit = "msSince1970">998403259909</timestamp>
    <value type = "base64">aG93iG5vDyBicm73biBjb3cNCg==</value>
  </data>
  <!-- array of floats -->
  <data name = "data_135">
    <propertyName>heatFlux</propertyName>
    <description>Heat flux through surface XY</description>
    <rasterName>surfaceXY</rasterName>
    <unit>J/(m*m)</unit>
    <value arrayType = "float[2,2]">
      0.1090000E-01 0.7400001E-02 0.9300001E-02 0.7000000E-02
    </value>
  </data>
  <!-- method call -->
  <methodCall name = "getProperty">
    <parameter name = "propertyName" intent = "in" type = "string">heatFlux</parameter>
```

```
  </methodCall>
  <!-- method response -->
  <methodResponse name = "thisIsNotAValidMethod">
    <fault>
      <faultCode>10</faultCode>
      <faultString>error: lookup for unknown function</faultString>
      <faultActor>m3.model.swim.xmlparse_expat.c</faultActor>
      <detail>thisIsNotAValidMethod</detail>
    </fault>
  </methodResponse>
  <!-- description of data without the data itself -->
  <data name = "data_135b">
    <propertyName>heatFlux</propertyName>
    <description>Heat flux through surface XY</description>
    <rasterName>surfaceXY</rasterName>
    <unit>J/(m*m)</unit>
    <readValue arrayType = "float[100,100]"/>
  </data>
</fhrgData>
```

3.4 Grid Job Definition Language (GJobDL)

Die GJobDL dient zur Beschreibung des dynamischen Verhaltens komplexer Grid-Anwendungen, die aus mehreren, miteinander gekoppelten Softwarekomponenten sowie deren Eingabe- und Ausgabedaten bestehen. Diese Beschreibung wird von der Grid-Architektur benötigt, um die Grid-Anwendung auf das eigentliche Computing-Grid abzubilden und um den Prozessablauf der Grid-Anwendung zu steuern.

Die GJobDL ist in zwei Teile gegliedert. Der erste Teil eines GJobDL-Dokuments enthält die GResourceDL-Beschreibung der für den Job benötigten Ressourcen gemäß Kapitel 3.1. Der zweite Teil der GJobDL enthält ein Modell des Prozessablaufs in Form eines Petrinetzes. Die GJobDL kann in einer späteren Version durchaus auch durch andere Methoden der Prozessmodellierung ergänzt werden, zunächst werden von der Grid-Architektur des Fraunhofer Resource Grids jedoch ausschließlich Ablaufmodelle in Form von Petrinetzen unterstützt.

In vielen Grid-Initiativen hat es sich durchgesetzt, den Prozessablauf von gekoppelten Anwendungen und Daten in Form eines *Directed Acyclic Graphs (DAG)* zu modellieren (siehe Abbildung 28). Ein Beispiel hierfür ist UNICORE, bei dem so genannte *AbstractJobs* auf der Basis von DAGs definiert werden [*Almond und Snelling*, 1999]. Andere Beispiele für die Verwendung von DAGs sind CONDOR [*Condor Team*, 2002] und das Projekt GriPhyN [*Deelman et al.*, 2002]. DAGs haben trotz ihrer großen Verbreitung aufgrund ihrer sehr einfachen Struktur mehrere wichtige Nachteile:

- Da sie „directed“ sind, kann mit ihnen keine beidseitige Kopplung zwischen Softwarekomponenten beschrieben werden.
- Sie sind „acyclic“, es können also keine Schleifen (while ... do) definiert werden.
- Sie beschreiben nur das Verhalten, nicht den Zustand des Systems.
- Es wird nur der Kontrollfluss, nicht jedoch der Datenfluss beschrieben.

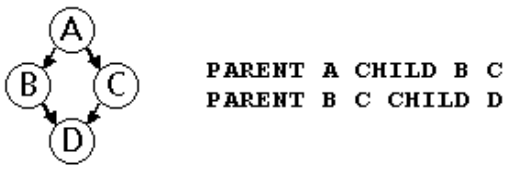

Abbildung 28: Beispiel eines *Directed Acyclic Graphs (DAG)*

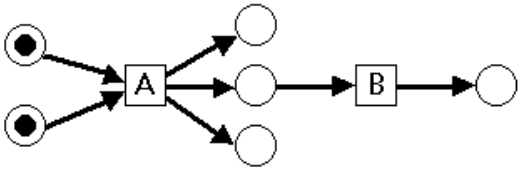

Abbildung 29: Beispiel eines *Petrinetzes*

Aus diesen Gründen sollen hier statt dessen Petrinetze zur Modellierung der Prozessabläufe verwendet werden (siehe Abbildung 29). Die Idee hierzu bekam ich durch Cezar Ionescu vom Potsdam-Institut für Klimafolgenforschung, der in seinem Prototyp eines „Graphical Simulation Builders“ einfache Petrinetze zur Ablaufsteuerung verwendet.

3.4.1 Petrinetze

Petrinetze erlauben es, sowohl den Zustand, als auch das dynamische Verhalten von Systemen zu beschreiben [*Petri*, 1962]. Die hier verwendete Prozessmodellierung entspricht vom Prinzip her dem Konzept eines Petrinetzes mit individuellen Marken (farbiges Petrinetz) und konstanten Pfeilanschriften, das nach *Reisig* [1985] gegeben ist durch

- *Stellen* (places), dargestellt als Kreise (◯),
- *Transitionen* (transitions), dargestellt als Kästchen (□),
- *Pfeile* (arcs) von Stellen zu Transitionen (◯ → □),
- *Pfeile* (arcs) von Transitionen zu Stellen (□ → ◯),
- individuelle, unterscheidbare *Objekte*, die als Marken (tokens) durch das Netz fließen können,
- eine *Anfangsmarkierung* (initial marking), die für jede Stelle festlegt, welche Objekte sie zu Beginn enthält und
- eine *Anschrift* an jedem Pfeil, die ein individuelles Objekt kennzeichnet.

Eine kurze Einführung in die theoretischen Aspekte von farbigen Petrinetzen ist zum Beispiel in *Jensen* [1994] zu finden.

In unserem Fall sollen Petrinetze nicht nur zur Modellierung von Prozessen, sondern darüber hinaus auch zur Ablaufsteuerung von Grid-Anwendungen verwendet werden. Der Kontrollfluss einer Grid-Anwendung lässt sich im Allgemeinen durch den Datenfluss zwischen den Softwarekomponenten darstellen, das heißt, die Entscheidung, wann eine Softwarekomponente ausgeführt werden soll, wird meist nur anhand der Verfügbarkeit der

notwendigen Eingabedaten gefällt. Die Marken des Petrinetzes repräsentieren somit die realen Daten, die zwischen den Softwarekomponenten im Grid ausgetauscht werden. Mit einem Petrinetz wird hier also das Zusammenspiel (= *wie*) zwischen Ressourcen vom Typ `software` und Ressourcen vom Typ `data` modelliert. Der Zusammenhang zwischen Softwarekomponenten und Hardwareressourcen, also *was* läuft *wo*, wird *nicht* im Petrinetz festgelegt, sondern erfolgt durch das Ressourcen-Mapping anhand der GResourceDL-Beschreibung (siehe Seite 13).

Ist der Kontrollfluss jedoch unabhängig vom Datenfluss zwischen den Softwarekomponenten, so werden zusätzliche *Kontrollstellen* und gegebenenfalls *Kontrolltransitionen* benötigt, deren Marken nur die Zustände von Prozessen repräsentieren (z. B. done, failed). Die Petrinetze werden daher für unsere Zwecke um folgende Eigenschaften erweitert:

- *Stellen* sind entweder *Datenstellen* oder *Kontrollstellen*. Datenstellen sind Referenzen auf Ressourcen vom Typ `data`. Kontrollstellen existieren hingegen nur „virtuell“ in der Grid-Architektur (z. B. im JobHandler).
- Stellen haben generell eine *Kapazität* von 1, das heißt, jede Stelle ist maximal mit einer einzigen Marke belegt.
- *Transitionen* sind entweder *Softwaretransitionen* oder *Kontrolltransitionen*. Softwaretransitionen sind Referenzen auf Ressourcen vom Typ `software`. Kontrolltransitionen existieren nur „virtuell“ in der Grid-Architektur und werden z. B. für die Auswertung logischer Bedingungen verwendet. Softwaretransitionen können eine Reihe von Eingabe- und Ausgabe-Ports besitzen.
- Ein *Pfeil* geht entweder von einer Stelle zu einem bestimmten *Port* einer Transition oder von einem bestimmten *Port* einer Transition zu einer Stelle.
- Die individuellen, unterscheidbaren *Objekte*, die als Marken (tokens) Datenstellen belegen, repräsentieren reale Daten, die z. B. in Dateien gespeichert sind.
- Die *Anschrift* eines Pfeils wird durch die Art des Ports der Transition festgelegt, der mit dem Pfeil verbunden ist.

Für Petrinetze mit individuellen Marken und konstanten Pfeilanschriften gilt außerdem folgendes (leicht abgewandelt nach *Reisig* [1985]):

- Eine Stelle s liegt im *Vorbereich* einer Transition t, wenn es einen Pfeil $\bigcirc \rightarrow \square$ von s nach t gibt (s = Eingabestelle von t).
- Eine Stelle s liegt im *Nachbereich* einer Transition t, wenn es einen Pfeil $\square \rightarrow \bigcirc$ von t nach s gibt (s = Ausgabestelle von t).
- Eine *Markierung* ist gegeben durch eine Verteilung von Objekten auf Stellen.
- Eine Transition t ist *aktiviert* (enabled), wenn
  1. jede Stelle s aus dem Vorbereich von t (Eingabestelle) das Objekt enthält, das die Anschrift des Pfeiles von s nach t bezeichnet und

2. jede Stelle im Nachbereich von t (Ausgabestelle) frei ist, das heißt, nicht mit einer Marke belegt ist.

- Eine aktivierte Transition t *schaltet* (fire), indem

  1. jeder Stelle s aus dem Vorbereich von t das Objekt entnommen wird, das der Pfeil von s nach t angibt, und
  2. jede Stelle s' aus dem Nachbereich von t das Objekt erhält, das der Pfeil von t nach s' angibt.

Geht ein Pfeil von einer Datenstelle zu einem Eingabeport einer Softwaretransition und ein weiterer Pfeil von einem Ausgabeport zu einer anderen Datenstelle, so bedeutet dies nach obigen Regeln Folgendes: Damit die Softwarekomponente, die durch die Transition repräsentiert wird, ausgeführt werden kann, müssen die Eingabedaten vorhanden sein, das heißt, die Eingabestelle muss mit einer Marke belegt sein. Aufgrund der *sicheren Transitionsregel*, die besagt, dass die Transition erst dann aktiviert ist, wenn auch die Ausgabestelle frei ist, kann eine Softwarekomponente erst dann ausgeführt werden, nachdem eine gegebenenfalls vorhandene Marke auf der Ausgabestelle durch das Schalten einer nachfolgenden Transition entfernt wurde.

Eigenschaften von Petrinetzen

Petrinetze können bestimmte Eigenschaften aufweisen, die mathematisch beschrieben werden können, und die es ermöglichen, Petrinetze in verschiedene Klassen zu unterteilen. Hier sollen die Begriffe nur umgangssprachlich eingeführt werden, die exakten mathematischen Formulierungen finden sich zum Beispiel in *Reisig* [1991], *van der Aalst et al.* [2001] sowie in *Gerber* [1999].

Arbeitsflussnetze (Workflow nets) bilden eine bestimmte Untermenge der Petrinetze, die nach *van der Aalst et al.* [2001] wie folgt definiert werden. Ein Petrinetz ist genau dann ein Arbeitsflussnetz, wenn gilt:

1. Es existiert eine *Quellstelle*, d.h. eine Stelle, die keine Vorgänger hat.
2. Es existiert eine *Senkstelle*, d.h. eine Stelle, die keine Nachfolger hat.
3. Jeder Knoten (=Teilmenge aus Stellen und Transitionen) liegt auf einem Pfad zwischen Quellstelle und Senkstelle.

Wir wollen hier zunächst die Menge der Petrinetze nicht auf reine Arbeitsflussnetze beschränken, da mittels der Petrinetze in unserem Fall nicht nur der Kontrollfluss, sondern auch der Datenfluss modelliert werden soll, wobei es durchaus mehrere verschiedene Quell- und Senkstellen geben kann.

Für den Begriff der *Lebendigkeit* von Stellen/Transitionsnetzen werden nach *Reisig* [1991] verschiedene Definitionen verwendet: Eine Markierung kann lebendig heißen,

1. wenn es unter jeder Folgemarkierung eine aktivierte Transition gibt,
2. wenn jede Transition einmal aktivierbar ist,
3. wenn jede Transition unter jeder Folgemarkierung aktivierbar ist,

4. wenn jede Folgemarkierung reproduzierbar ist,
5. wenn mindestens eine Folgemarkierung reproduzierbar ist
6. ...

Ein Netz heißt *lebendig*, wenn es bezüglich obiger Lebendigkeitsbegriffe für Markierungen lebendig markierbar ist. Auf die Modellierung von Grid-Anwendungen übertragen, würde Definition eins in der Regel nicht erfüllt sein, da analog zu den Arbeitsflussnetzen nach erfolgreicher Ausführung einer Grid-Anwendung keine der Transitionen aktiviert ist. Die verbleibenden Marken liegen oftmals zum Schluss auf Stellen, die keine nachfolgenden Transitionen haben (siehe zum Beispiel Abbildung 38). Da für uns die Lebendigkeit des Netzes der Lebendigkeit einer Grid-Anwendung gleichkommt, ist die zweite Definition des Begriffs Lebendigkeit die zutreffendste. Ich schlage folgende Definition vor:

> Ein Petrinetz, welches eine Grid-Anwendung modelliert, heißt *lebendig*, wenn bei der gegebenen Anfangsmarkierung jede der Transitionen mindestens einmal aktivierbar ist.

Ein Petrinetz, das Transitionen beinhaltet, die nie aktiviert werden können, ist in diesem Sinne also nicht lebendig. Petrinetze, die nicht lebendig sind, enthalten zum Beispiel überflüssige oder nicht erreichbare Transitionen oder haben eine unzureichende Anfangsmarkierung.

Falls zwei Transitionen aktiviert sind und durch das Schalten der einen Transition die andere nicht mehr aktiviert ist, so spricht man von einem *Konflikt* [*Reisig*, 1985]. Konflikte können zum Beispiel auftreten, wenn sich mehrere Transitionen die selbe Eingabestelle teilen. In manchen Fällen ist es unklar, ob Konflikte eintreten. Solche Situationen nennt man *konfus*.

Unter einer Markierung M liegt bei einer Transition t ein *Kontakt* vor, wenn t nur wegen zu geringer Kapazität einer Ausgabestelle nicht schalten kann. Da bei uns die Kapazität einer jeden Stelle 1 beträgt, ist dies immer dann der Fall, wenn zwar alle Eingabestellen mit den richtigen Objekten belegt sind, aber mindestens eine der Ausgabestellen nicht frei ist. Kontakte können durch die Konstruktion von *Komplementen* vermieden werden (siehe *Reisig* [1985]). Kontakte können in unserem Fall durchaus gewollt sein, zum Beispiel, um das vorzeitige Überschreiben von Daten, die noch nicht von einer nachfolgenden Softwarekomponente weiterverarbeitet wurden, zu verhindern.

Kommen wir nun zu der Definition der Begriffe *sauber, markiert, Falle* und *Deadlock* (nach *Gerber* [1999]):

- Falls keine der Stellen Q (Q ist Teilmenge aller Stellen S) bei einer bestimmten Markierung M markiert ist, nennt man Q *sauber* bei M, ansonsten heißt Q *markiert*.
- Q wird *Falle* genannt, wenn Q markiert ist und nicht gesäubert werden kann.
- Q heißt *Deadlock*, wenn ein sauberes Q nicht markiert werden kann.

Eine für die Handhabung von Petrinetzen sehr praktische Eigenschaft ist die Möglichkeit des *Verfeinerns* und *Einbettens*. Man verfeinert ein Netz, indem man eine Stelle oder eine Transition durch ein ganzes Netz ersetzt und das Resultat wieder ein Netz ergibt. Nach *Reisig* [1985] gilt dabei Folgendes:

In einem Netz A wird eine Stelle s durch das Netz B verfeinert, wenn B anstelle von s so eingesetzt wird, dass für jeden Pfeil $x \rightarrow y$ von B nach A' (bzw. $x \leftarrow y$ von A' nach B) gilt:

- x ist eine Stelle von B und y ist eine Transition von A';
- in A gibt es einen Pfeil $s \rightarrow y$ (bzw. $s \leftarrow y$).

A' bezeichnet dabei das Netz A ohne die Stelle s.

Analoges gilt für die Verfeinerung von Transitionen. Beim *Einbetten* wird nicht nur eine einzelne Stelle oder Transition ersetzt, sondern das gegebene Netz wird durch ein neues Netzteil ergänzt. Die zusätzliche Konstruktion des Komplements von Stellen wäre eine solche Einbettung.

Prozessmodellierung mit Petrinetzen

Mit Hilfe von Petrinetzen lassen sich alle diskreten Prozessabläufe auf einfache Art und Weise mit den drei Grundbausteinen — Stellen, Transistionen und Pfeilen — modellieren. Eine Übersicht über die Darstellung von Prozessmodellen mittels Petrinetzen ist in *van der Aalst und Kumar* [2000] zu finden. Hieraus sind auch die folgenden Abbildungen entnommen.

Jede einfache Aufgabe (task), die in einem Petrinetz durch eine Transition repräsentiert wird, läßt sich wiederum nach den Prinzipien der Netzverfeinerung durch ein eigenes Petrinetz darstellen. Mehrere dieser einzelnen Aufgaben lassen sich nun zum Beispiel sequentiell (zeitlich aufeinanderfolgend) oder wahlweise ausführen (siehe Abbildung 30) oder durch Bedingungen miteinander verknüpfen (siehe Abbildung 31). Mittels Petrinetzen können insbesondere auch Schleifen (siehe Abbildung 32) sowie paralleles Ausführen, sowohl mit (siehe Abbildung 33), als auch ohne anschließende Synchronisation (siehe Abbildung 34) modelliert werden. Desweiteren kann die Ausführung von Aufgaben auch von bestimmten Ereignissen abhängen (siehe Abbildungen 35 und 36).

Sollen Prozesse nicht nur abstrakt modelliert, sondern auch deren dynamisches Verhalten konkret simuliert werden, so werden einige spezielle Transitionen benötigt, die insbesondere bei Wahlmöglichkeiten im Petrinetz zwischen den einzelnen möglichen Wegen entscheiden. Eine Marke kann immer dann einem von mehreren möglichen Wegen folgen, wenn von einer Stelle aus mehrere Pfeile zu verschiedenen Transitionen hinführen und die Transitionen nach obiger Definition in einem Konflikt zueinander stehen. Es kann nur eine der konkurrierenden Transitionen schalten, auch wenn mehrere aufgrund der belegten Eingabestelle aktiviert sind. Welche Transition schaltet, wird dabei nicht direkt durch das Petrinetz vorgegeben. In den abgebildeten Petrinetzen sind es insbesondere die Transitionen *true, false* und *time_out*, die nur unter bestimmten Umständen schalten, also nicht zwangsweise nachdem sie aktiviert worden sind. Die Transitionen *true* und *false* werten bestimmte Daten aus und entscheiden danach, ob sie schalten. Die Transition *time_out* schaltet erst, nachdem sie eine festgelegte Zeitspanne lang ununterbrochen aktiviert war. Die Transitionen *begin, end, AND-split* und *AND-join* schalten hingegen direkt nachdem sie aktiviert worden sind, da sie keine Wahlmöglichkeit für Marken darstellen.

Für die Verwendung von Petrinetzen im Fraunhofer Resource Grid wurden weiter oben im Text einige Erweiterungen der Petrinetze vorgenommen. Wesentliche Änderungen sind

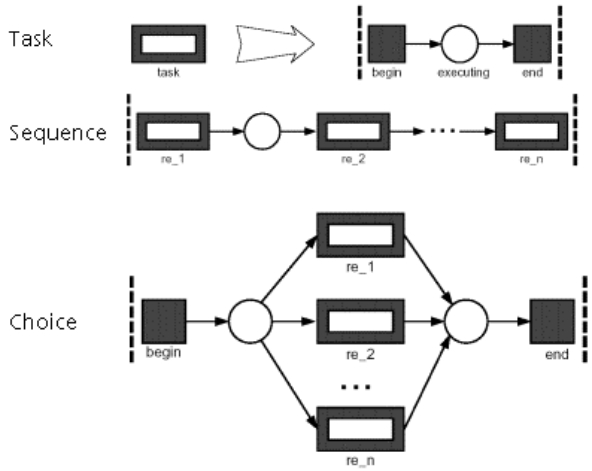

Abbildung 30: Modellierung einer *Aufgabe (task)*, eines *sequentiellen Ablaufs (sequence)* und einer *Wahl (choice)* mit Hilfe von Petrinetzen nach *van der Aalst und Kumar* [2000].

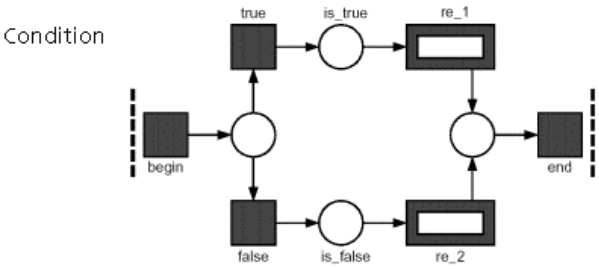

Abbildung 31: Modellierung einer *Bedingung (condition)* mittels eines Petrinetzes nach *van der Aalst und Kumar* [2000].

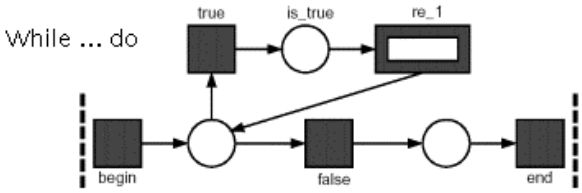

Abbildung 32: Modellierung einer *While-Do-Schleife* mittels eines Petrinetzes nach *van der Aalst und Kumar* [2000].

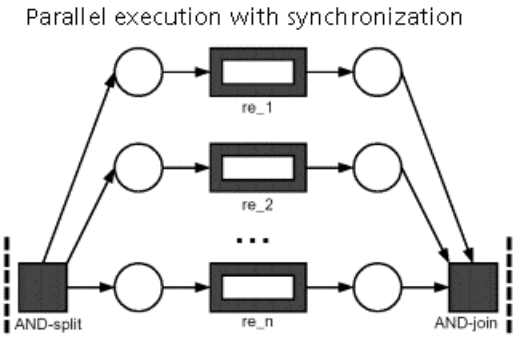

Abbildung 33: Modellierung einer *parallelen Ausführung mit Synchronisation* mittels eines Petrinetzes nach *van der Aalst und Kumar* [2000].

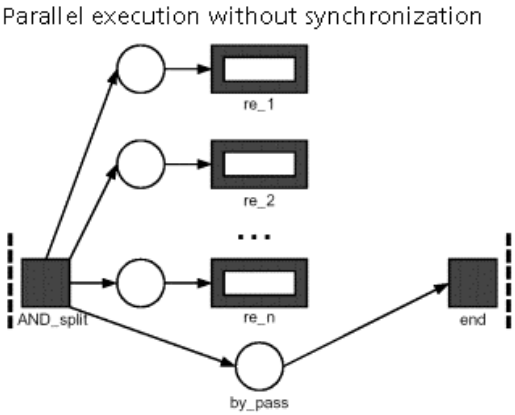

Abbildung 34: Modellierung einer *parallelen Ausführung ohne Synchronisation* mittels eines Petrinetzes nach *van der Aalst und Kumar* [2000].

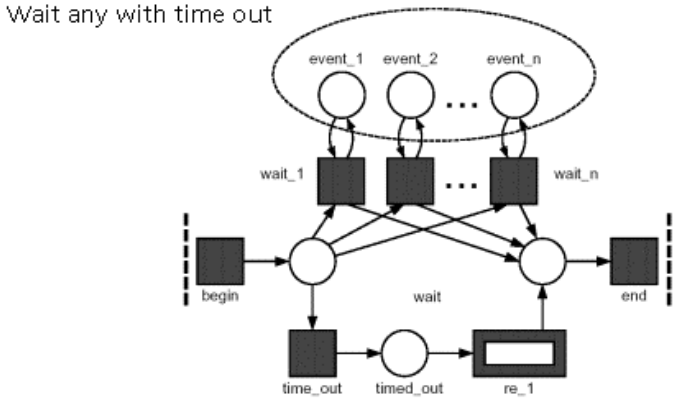

Abbildung 35: Modellierung eines *wait any with time out* mittels eines Petrinetzes nach *van der Aalst und Kumar* [2000].

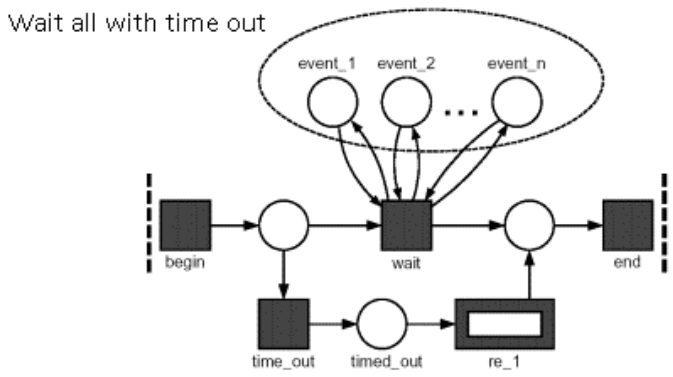

Abbildung 36: Modellierung eines *wait all with time out* mittels eines Petrinetzes nach *van der Aalst und Kumar* [2000].

die Einführung so genannter *Ports* und die Unterteilung der Transitionen und Stellen in Elemente, die ausschließlich den Kontrollfluss modellieren, sowie in Elemente, die zudem zur Modellierung des Datenflusses dienen. In Abbildung 37 sind diese Elemente mit einem Vorschlag einer grafischen Repräsentation abgebildet. Abbildung 38 zeigt ein Beispiel der Petrinetz-Modellierung einer Grid-Anwendung, die mit Hilfe des Linux-Programms „cat“ mehrere Dateien zu einer neuen zusammenfügt.

Wird mit Hilfe eines Pfeils eine Datenstelle mit einem Port einer Softwaretransition verbunden, so beinhaltet das zwei Informationen:

1. **Kontrollfluss:** Erst wenn alle Eingabestellen markiert sind, ist eine Softwaretransition aktiviert und darf schalten, d. h. ausgeführt werden.

2. **Datenfluss:** Wenn es sich bei der Stelle um eine Datenstelle handelt, so ist der Inhalt der Stelle (z. B. der Inhalt einer Datei) an den entsprechenden Port (stdin, Socket, Datei) der Softwaretransition zu übermitteln.

Anforderungen an Softwarekomponenten und Daten

Damit gekoppelte Grid-Anwendungen mit Hilfe der hier vorgestellten Petrinetze modelliert werden können und die Ausführung der Einzelanwendungen sowie deren Verknüpfung mit den Daten anhand des Prozessmodells gesteuert werden können, müssen einige Anforderungen an die auszuführenden Softwarekomponenten und die zu verarbeitenden Daten gestellt werden:

- **Softwarekomponenten:** Eine Softwarekomponente, die als einzelne Transition repräsentiert wird, kann als Blackbox mit wohldefinierten Eingabe- und Ausgabeports beschrieben werden. Beim Schalten (Ausführen) werden die Eingabedaten gelesen und die Ausgabedaten ausgegeben. Das Schalten eines Programms erfolgt in standardisierter Form (zum Beispiel durch das Submitten des Programms per GRAM).

- **Daten:** Es gibt einen Mechanismus, über den festgestellt werden kann, ob die Eingabedaten vollständig vorhanden sind (= Eingabestelle ist mit Marke belegt).

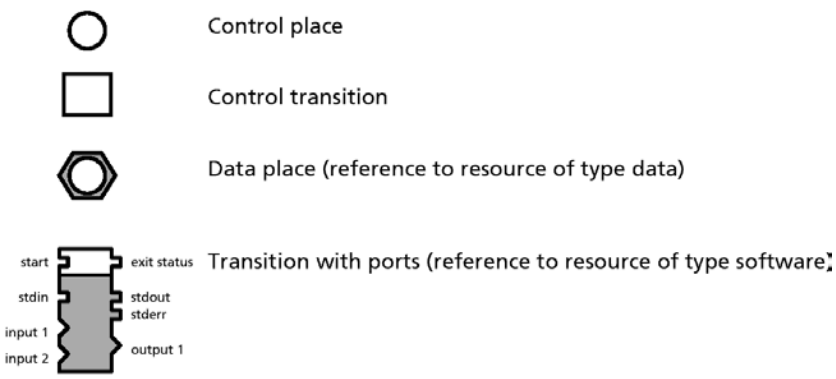

Abbildung 37: Die Elemente eines erweiterten Petrinetzes zur Modellierung des Kontroll- und Datenflusses im Fraunhofer Resource Grid. Von oben nach unten: Kontrollstelle (control place), Kontrolltransition (control transition), Datenstelle (data place) und eine Softwaretransition mit mehreren Eingabe- und Ausgabeports.

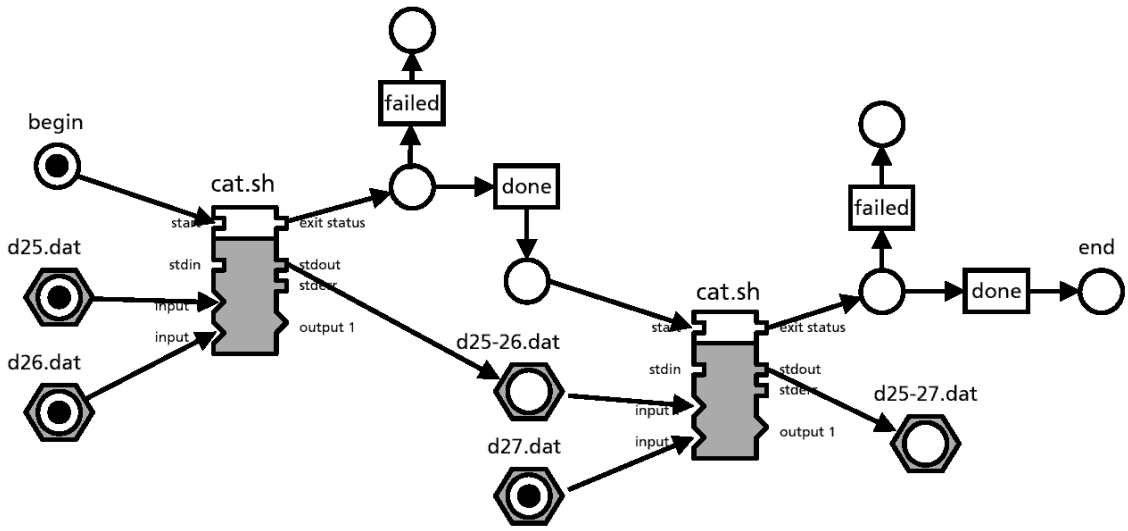

Abbildung 38: Beispiel eines Petrinetzes einer gekoppelten Grid-Anwendung, welche in zwei Schritten die Dateien *d25.dat, d26.dat* und *d27.dat* zu einer neuen Datei *d25-27.dat* zusammenfügt.

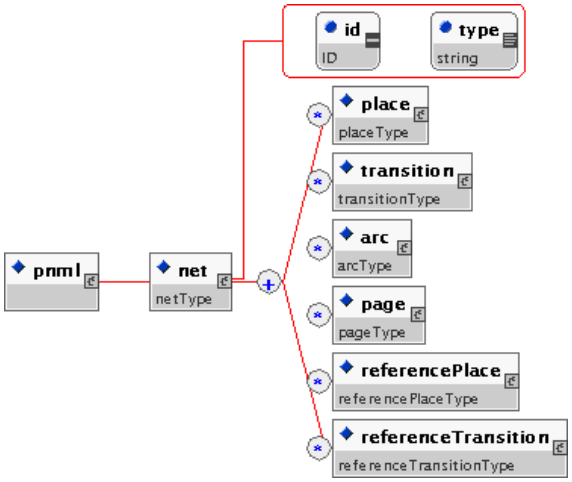

Abbildung 39: Die Grundelemente der *Petri Net Markup Language (PNML)* nach *Jüngel et al.* [2000] und *Kindler und Weber* [2001].

Auf die Daten kann in standardisierter Form zugegriffen werden (zum Beispiel per GridFTP).

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

Petrinetze und XML

Es existieren bereits mehrere Ansätze zur Beschreibung von Petrinetzen mit Hilfe von XML. Am verbreitetsten ist die *Petri Net Markup Language*, die am Institut für Informatik der Humboldt-Universität zu Berlin entwickelt wurde [*Kindler und Weber*, 2001; *Jüngel et al.*, 2000]. Mit Hilfe des *Petri Net Kernels* (http://www.informatik.hu-berlin.de/top/pnk/index.html) können in PNML beschriebene Petrinetze gelesen und grafisch ausgewertet sowie simuliert werden. Neben dem mathematischen Inhalt der Petrinetze werden mit der PNML auch Informationen gespeichert, die zur Visualisierung der Petrinetze benötigt werden, also zum Beispiel die X/Y-Koordinaten und Beschriftungen der verschiedenen Elemente. Abbildung 39 zeigt einige Grundelemente der PNML.

Ein zur PNML sehr ähnlicher Ansatz zur Beschreibung von Petrinetzen mit XML — jedoch mit einer etwas anderen Syntax — wird in *Mailund und Mortensen* [2000] vorgestellt, wobei hier versucht wird, den Inhalt und das Format der Petrinetze zu trennen und auf unterschiedliche Dokumente zu verteilen.

Beide der soeben vorgestellten Beschreibungssprachen erfüllen jedoch nicht die gegebenen Anforderung für die Verwendung im Rahmen des Fraunhofer Resource Grids, sodass für die GJobDL eine eigene, an die PNML angelehnte Beschreibungssprache für Petrinetze verwendet wird. Insbesondere fehlt bei der PNML die Möglichkeit, Softwaretransitionen und Datenstellen mit real existierenden Ressourcen zu verknüpfen und den Datenaustausch über Ports zu beschreiben. Es sollte jedoch leicht möglich sein, GJobDL-Dokumente mittels XML-Stylesheets (XSLT) in PNML-Dokumente umzuwandeln. Im folgenden Kapitel wird das Datenmodell der GJobDL mit der darin enthaltenen Petrinetzbeschreibung vorgestellt.

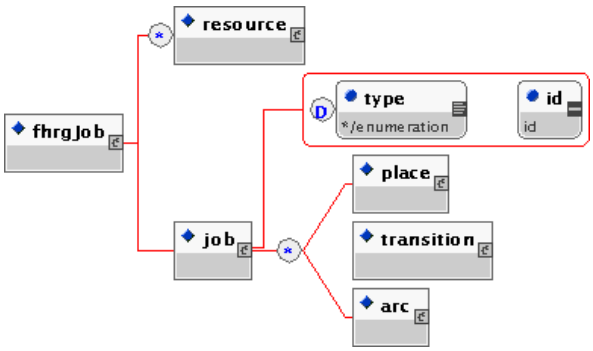

Abbildung 40: Die Grundelemente der GJobDL. Das Element *resource* ist identisch mit dem gleichnamigen Element aus der GResourceDL. Das Element *job* dient zur Beschreibung des Kontroll- und Datenflusses der Grid-Anwendung mit Hilfe eines Petrinetzes, bestehend aus Stellen (*place*), Transitionen (*transition*) und Pfeilen (*arc*).

3.4.2 Datenmodell

```
<fhrgJob>

(resource* , job)
```

Ein GJobDL-Dokument besteht aus zwei Teilen. Im ersten Teil können null oder mehrere Ressourcenbeschreibungen der an der Grid-Anwendung beteiligten Ressourcen mit Hilfe des Elements *resource* aufgelistet werden. Anschließend wird im zweiten Teil mittels des Elements *job* der Kontroll- und Datenfluss der Grid-Anwendung definiert (siehe Abbildung 40).

```
<resource type="..." id="...">
```

Der Elementtyp *resource* ist identisch mit dem gleichnamigen Elementtyp aus der GResourceDL (siehe Kapitel 3.1.2).

```
<job type="petriNet" id="...">

(place | transition | arc)*
```

Mit Hilfe diesen Elementtyps wird der Kontroll- und Datenfluss der Grid-Anwendung beschrieben (Abbildung 40). Zur Zeit werden als einzig mögliche Art der Job-Modellierung Petrinetze unterstützt, was durch das Attribut `type="petriNet"` festgelegt wird. Das

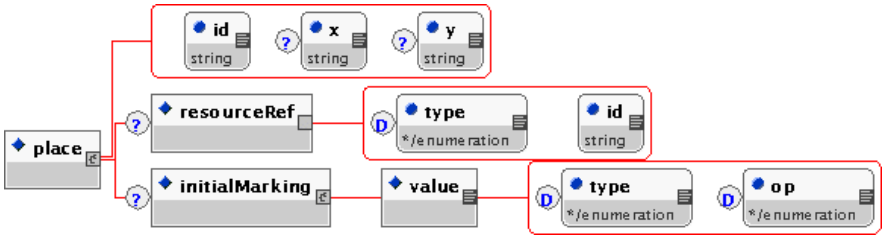

Abbildung 41: Das Element *place* der GJobDL mit Unterelementen, das zur Beschreibung der Stellen eines Petrinetzes dient.

Attribut `id` dient zur Identifikation, das heißt zur eindeutigen Kennzeichnung der Job-Beschreibung. Diese Kennzeichnung kann zum Beispiel analog zum Attribut `id` des Elements `resource` der GResourceDL erfolgen. Statt den zwei Buchstaben (SW, SC, HW, HC, DA, DC) für den Ressourcentyp beginnt die Kennung von Job-Beschreibungen mit dem Wort `JOB`. Eine vollständige Kennung wäre dann zum Beispiel:

```
JOB02_000002_de-fhrg-first_concatenateIt
```

Mögliche Unterelemente von `job` sind die drei Grundelemente eines Petrinetzes: Stellen (`place`), Transitionen (`transition`) und Pfeile (`arc`).

```
<place id="..." x="..." y="...">
(resourceRef? , initialMarking?)
```

Der Elementtyp *place* dient zur Beschreibung der Stellen eines Petrinetzes (Abbildung 41). Eine Stelle kann entweder eine Kontrollstelle oder eine Datenstelle darstellen. Für den Fall einer Datenstelle wird die Stelle mit einer Ressource vom Typ `data` mit Hilfe des Elementtyps `resourceRef` verknüpft. Das Attribut `id` des Elements `resourceRef` muss dann mit der Kennung der verknüpften Ressource exakt übereinstimmen. Das Attribut `id` des Elements `place` muss für jedes GJobDL-Dokument eindeutig sein. Die Attribute `x` und `y` dienen zur Angabe der Koordinate einer Stelle für deren Visualisierung, zum Beispiel im JobBuilder. Mit dem Elementtyp `initialMarking` kann gegebenenfalls ein Objekt als Anfangsmarkierung für diese Stelle definiert werden. Ohne die Angabe einer Anfangsmarkierung wird angenommen, dass zu Beginn die Stelle nicht von einer Marke belegt ist.

```
<transition id="..." x="..." y="...">
(resourceRef | condition)
```

Mit *transition* werden die Transitionen eines Petrinetzes beschrieben (Abbildung 42). Das Attribut `id` dient zur eindeutigen Identifizierung der Transition innerhalb eines GJobDL-Dokuments. Mit den Attributen `x` und `y` kann die Koordinate der Transition angegeben werden. Transitionen können entweder Kontrolltransitionen oder Softwaretransitionen darstellen. Handelt es sich um eine Softwaretransition, so wird mittels des Elements

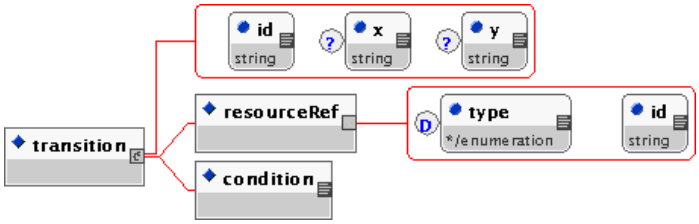

Abbildung 42: Das Element *transition* der GJobDL mit Unterelementen, das zur Beschreibung der Transitionen eines Petrinetzes dient.

`resourceRef` die Transition mit einer Ressource vom Typ `software` verknüpft. Falls die Transition eine Kontrolltransition darstellt, so kann mittels des Elements `condition` eine Bedingung angegeben werden, die erfüllt sein muss, damit die aktivierte Transition schaltet. Die Syntax von `condition` ist gleich der Java-Syntax. Die Methodenaufrufe, die hier verwendet werden können, sind durch eine spezielle Schnittstelle, die im JobHandler implementiert wird, festgelegt. Zur Zeit sind folgende Bedingungen vorgesehen:

```
isFailed()
isDone()
```

Die Funktionsweise dieser Bedingungen ist wie folgt zu verstehen: Nachdem eine Softwaretransition geschaltet hat (d.h. ausgeführt wurde), wird deren Exit-Status in einem Objekt gespeichert und als Marke auf alle Kontrollstellen im Nachbereich gesetzt. Folgt dieser Kontrollstelle nun eine Kontrolltransition mit der Bedingung `isFailed()`, so schaltet diese Transition nur dann, wenn der in der Marke gespeicherte Exit-Status dem Wert `failed` entspricht und die Transition aktiviert ist. Eine Kontrolltransition mit der Bedingung `isDone()` würde hingegen nur dann schalten, wenn der Exit-Status dem Wert `done` entspricht. Die Liste der möglichen Bedingungen soll im Laufe der Zeit um weitere Funktionalitäten, wie zum Beispiel Zähler, Vergleich mit Zeit/Datum etc. ergänzt werden. Ergibt die Bedingung den Wert `true` oder wird keine Bedingung angegeben, so schaltet die aktivierte Kontrolltransition.

```
<arc type="..." id="...">

((placeRef , transitionRef) | (transitionRef , placeRef))
```

Mit Hilfe des Elementtyps *arc* können Pfeile definiert werden, die entweder Stellen mit Transitionen (`type="P2T"`) oder Transitionen mit Stellen (`type="T2P"`) verbinden (Abbildung 43). Handelt es sich bei der Transition um eine Softwarestelle, so kann der Pfeil auch mit einem bestimmten Port der Softwaretransition verknüpft werden (siehe nachfolgenden Elementtyp `transitionRef`). In diesem Fall wird neben dem Kontrollfluss auch ein Datenfluss impliziert. Der Wert des Attributs `id` muss eine eindeutige Identifizierung des Pfeils in einem GJobDL-Dokument ermöglichen.

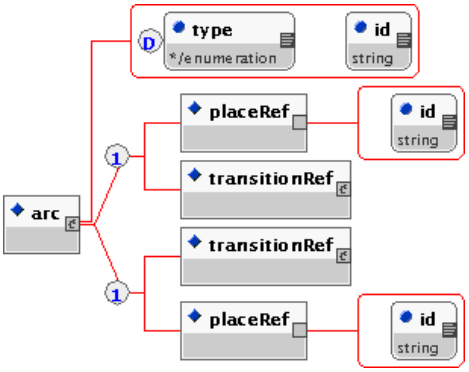

Abbildung 43: Das Element *arc* der GJobDL mit Unterelementen, das zur Beschreibung der Pfeile eines Petrinetzes dient.

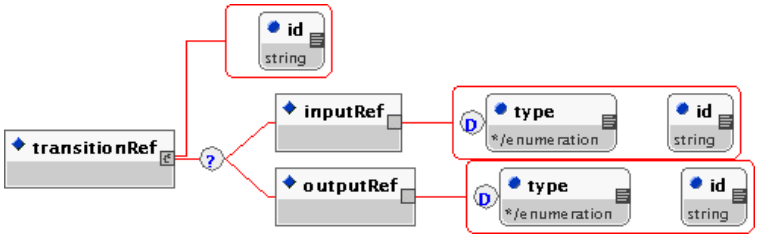

Abbildung 44: Das Element *transitionRef* der GJobDL, das entweder eine Referenz auf eine Transition oder die Referenz auf einen bestimmten Eingabe- bzw. Ausgabeport einer Softwaretransition enthält.

```
<transitionRef id="...">

(inputRef | outputRef)?
```

Mit dem Elementtyp *transitionRef* kann entweder auf eine Transition oder auf einen speziellen Port einer Softwaretransition verwiesen werden. Soll auf einen speziellen Port verwiesen werden, so sind die Elementtypen `inputRef` für die Angabe des Eingabeports und `outputRef` für die Angabe des Ausgabeports zu verwenden. Die Werte der Attribute `type` und `id` von `inputRef` bzw. `outputRef` stimmen mit dem Typ und der Kennung der Eingabe- bzw. Ausgabeports der GResourceDL überein (siehe Kapitel 3.1.2). Kontrolltransitionen besitzen keine Ports. Falls auf eine Softwaretransition ohne die Angabe eines Eingabe- bzw. Ausgabeports verwiesen wird, so wird standardmäßig nur der Kontrollfluss der Softwaretransition berücksichtigt. Pfeile, die auf diese Weise ohne die Angabe einer `inputRef` an einer Softwaretransition enden, sind somit mit `start` verbunden. Pfeile, die ohne die Angabe einer `outputRef` an einer Softwaretransition beginnen, sind mit `exit status` verbunden (siehe Abbildung 38 auf Seite 55).

3.4.3 Beispiele

Als Beispiel eines GJobDL-Dokuments sollen hier Auszüge aus der Beschreibung der Grid-Anwendung aus Abbildung 38 dienen. Im ersten Teil des GJobDL-Dokuments werden optional die aus der GResourceDL übernommenen Definitionen der beteiligten Ressourcen vorgenommen. Fehlt dieser Teil oder werden nicht alle beteiligten Ressourcen in dem GJobDL-Dokument aufgelistet, so muss es eine Möglichkeit geben, auf anderem Wege die Ressourcenbeschreibungen zu erhalten, zum Beispiel durch Abfrage eines Repositorys, das von der Grid-Architektur zur Verfügung gestellt wird.

```
<?xml version = "1.0" encoding = "ISO-8859-1"?>
<!DOCTYPE fhrgJob SYSTEM "gjdl0_2.dtd">
<fhrgJob>
  <resource id = "SW02_000002_de-fhrg-first_cat" type = "software">
    [...]
    <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>
    [...]
    <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>/bin</directory>
      <filename>cat</filename>
    </location>
    [...]
    <softwareSpecific.static>
      <output id = "stdout" type = "stdout"/>
      <output id = "stderr" type = "stderr"/>
      [...]
    </softwareSpecific.static>
    <softwareSpecific.variable>
      <input id = "input1" type = "file"/>
      <input id = "input2" type = "file"/>
    </softwareSpecific.variable>
  </resource>

  <resource id = "DA02_000001_de-fhrg-first_d25" type = "data">
    [...]
  </resource>

  <resource id = "DA02_000001_de-fhrg-first_d26" type = "data">
    [...]
  </resource>
```

```
<resource id = "DA02_000001_de-fhrg-first_d27" type = "data">
  [...]
</resource>

<resource id = "DA02_000001_de-fhrg-first_d25-26" type = "data">
  [...]
</resource>

<resource id = "DA02_000001_de-fhrg-first_d25-27" type = "data">
  [...]
</resource>

[...]
```

Der zweite Teil des GJobDL-Dokuments enthält die Definition des Kontroll- und Datenflusses in Form eines Petrinetzes mit der Zuordnung der entsprechenden Software- und Datenressourcen:

```
<job type = "petriNet" id = "JOB02_000002_de-fhrg-first_concatenateIt">
  <place id = "p_begin">
    <initialMarking>
      <value type = "boolean" op = "eq">true</value>
    </initialMarking>
  </place>
  <place id = "d25">
    <resourceRef id = "DA02_000001_de-fhrg-first_d25" type = "data"/>
    <initialMarking>
      <value type = "boolean" op = "eq">true</value>
    </initialMarking>
  </place>
  <place id = "d26">
    <resourceRef id = "DA02_000001_de-fhrg-first_d26" type = "data"/>
    <initialMarking>
      <value type = "boolean" op = "eq">true</value>
    </initialMarking>
  </place>
  <place id = "d27">
    <resourceRef id = "DA02_000001_de-fhrg-first_d27" type = "data"/>
    <initialMarking>
      <value type = "boolean" op = "eq">true</value>
    </initialMarking>
  </place>
  <place id = "d25-26">
    <resourceRef id = "DA02_000001_de-fhrg-first_d25-26" type = "data"/>
  </place>
  <place id = "d25-27">
    <resourceRef id = "DA02_000001_de-fhrg-first_d25-27" type = "data"/>
  </place>
  <place id = "p_exitStatus1"/>
  <place id = "p_done1"/>
  <place id = "p_failed1"/>
  <place id = "p_exitStatus2"/>
  <place id = "p_failed2"/>
  <place id = "p_end"/>

  <transition id = "t_cat1">
```

```
      <resourceRef type = "software" id = "SW02_000002_de-fhrg-first_cat"/>
    </transition>
    <transition id = "t_cat2">
      <resourceRef type = "software" id = "SW02_000002_de-fhrg-first_cat"/>
    </transition>
    <transition id = "t_failed1">
      <condition>isFailed()</condition>
    </transition>
    <transition id = "t_done1">
      <condition>isDone()</condition>
    </transition>
    <transition id = "t_failed2">
      <condition>isFailed()</condition>
    </transition>
    <transition id = "t_done2">
      <condition>isDone()</condition>
    </transition>

    <arc id = "arc1" type = "P2T">
      <placeRef id = "p_begin"/>
      <transitionRef id = "t_cat1"/>
    </arc>
    <arc id = "arc2" type = "P2T">
      <placeRef id = "d25"/>
      <transitionRef id = "t_cat1">
        <inputRef id = "input1" type = "file"/>
      </transitionRef>
    </arc>
    <arc id = "arc3" type = "P2T">
      <placeRef id = "d26"/>
      <transitionRef id = "t_cat1">
        <inputRef id = "input2" type = "file"/>
      </transitionRef>
    </arc>
    <arc id = "arc4" type = "T2P">
      <transitionRef id = "t_cat1"/>
      <placeRef id = "p_exitStatus1"/>
    </arc>
    <arc id = "arc5" type = "T2P">
      <transitionRef id = "t_cat1">
        <outputRef id = "stdout" type = "stdout"/>
      </transitionRef>
      <placeRef id = "d25-26"/>
    </arc>

    [...]

  </job>
</fhrgJob>
```

4 Ausblick

Diese Version der GADL stellt einen Prototyp dar, dessen Anwendbarkeit in der Praxis noch durch verschiedene Beispielanwendungen getestet werden muss. Als Beispielanwendungen im Rahmen des Fraunhofer Resource Grids sind zunächst von Fraunhofer

ITWM ein bis zwei gekoppelte Grid-Anwendungen, voraussichtlich aus dem Bereich der Mikrostruktursimulation geplant. Fraunhofer FIRST arbeitet zur Zeit an einer gekoppelten Simulation zur Ausbreitung von Schadstoffen in der Luft, im Boden und im Wasser, die als weiteres Beispiel einer Grid-Anwendung dienen soll.

Ein nächster wichtiger Schritt ist die Einbindung der GADL in das Repository, in den JobBuilder und in den JobHandler des Fraunhofer Resource Grids. Der JobHandler soll hierfür stufenweise ausgebaut werden, beginnend bei einer einfachen Implementierung der Steuerung des reinen Kontrollflusses per Petrinetz. Insbesondere die Liste möglicher Bedingungen von Kontrolltransitionen wird im Laufe der Zeit erweitert werden müssen, um auch komplexere Zusammenhänge zwischen den einzelnen Softwarekomponenten und deren Daten darstellen zu können.

Durch die Verwendung von Petrinetzen zur Modellierung der Kontroll- und Datenflüsse kann auf einen großen Fundus mathematischer Hilfsmittel zugegriffen werden, der für die Analyse und Optimierung der Netze eingesetzt werden kann. Insbesondere die Untersuchung der Netze auf Lebendigkeit, Fallen und Deadlocks scheint für unsere Anwendung der Petrinetze sehr erfolgversprechend zu sein. Ziel sollte es also sein, diese mathematischen Hilfsmittel möglichst bald in die Grid-Architektur zu integrieren.

Es ist zudem geplant, eine Kurzfassung dieses Textes in Englisch zu erstellen.

Literatur

Almond, J. und D. Snelling, UNICORE: secure and uniform access to distributed resources via the world wide web, 1999.

Anderson, R., *XML professionell*, MITP-Verlag, Bonn, 2000.

Apache, SOAP, http://xml.apache.org/soap/, 2001.

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.

Bray, T., J. Paoli, C. M. Sperberg-McQueen und E. Maler, Extensible Markup Language (XML) 1.0 — W3C Recommendation 6 October 2000, technical report, World Wide Web Consortium (W3C), http://www.w3.org/TR/2000/REC-xml-20001006, 2000.

Christensen, E., F. Curbera, G. Meredith und S. Weerawarana, Web Services Description Language (WSDL) 1.1, technical report, World Wide Web Consortium (W3C), http://www.w3.org/TR/wsdl, 2001.

Condor Team, Condor Version 6.4.0 Manual, http://www.cs.wisc.edu/condor/ downloads/index.html, 2002.

Deelman, E., C. Kesselman, G. Mehta, L. Meshkat, L. Pearlman, K. Blackburn, P. Ehrens, A. Lazzarini, R. Williams und S. Koranda, GriPhyN and LIGO, building a virtual data grid for gravitational wave scientists, in *Proceedings of*

the eleventh IEEE International Symposium on High Performance Distributed Computing, 23-16 July 2002, 225–234, Edinburgh, Scotland, 2002.

Freed, N. und N. Borenstein, RFC2045: Multipurpose Internet Mail Extensions (MIME) — Part One: Format of Internet Message Bodies, technical report, Innosoft, First Virtual, http://www.normos.org/ietf/rfc/rfc2045.txt, 1996.

Gerber, S., Vorlesungsskript WS 1999/2000: Petri-Netze, http://www.informatik.uni-leipzig.de/~gerber/petri.ps, 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.

Jensen, K., An introduction to the theoretical aspects of Coloured Petri Nets, in *A Decade of Concurrency, Lecture Notes in Computer Science*, herausgegeben von J. W. de Bakker, W.-P. de Roever, und G. Rozenberg, Bd. 803, 230–272, Springer-Verlag, http://www.daimi.au.dk/PB/476/PB-476.pdf, 1994.

Jüngel, M., E. Kindler und M. Weber, The Petri Net Markup Language, in *7. Workshop Algorithmen und Werkzeuge für Petrinetze*, herausgegeben von S. Philippi, 47–52, Universität Koblenz-Landau, http://www.informatik.hu-berlin.de/top/pnml/, 2000.

Kindler, E. und M. Weber, A universal module concept for Petri nets. An implementation-oriented approach, technical report, http://www.informatik.hu-berlin.de/top/pnml/, 2001.

Mailund, T. und K. Mortensen, Separation of style and content with XML in an interchange format for high-level Petri nets, in *Proceedings of the Meeting on XML/SGML based Interchange Formats for Petri Nets*, 7–12, Aarhus, Denmark, 2000.

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

Petri, C. A., *Kommunikation mit Automaten*, Dissertation, Institut für Instrumentelle Mathematik, Bonn, 1962.

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.

Reisig, W., *Systementwurf mit Netzen*, Springer-Verlag, 1985.

Reisig, W., *Petrinetze — Eine Einführung*, Springer-Verlag, 2 Auflage, 1991.

Sun Microsystems, Code conventions for the java programming language, http://java.sun.com/docs/codeconv/, 1999.

The Globus Project, GRAM RSL Parameters, http://www.globus.org/gram/gram_rsl_parameters.html, 2000*a*.

The Globus Project, The Globus Resource Specification Language RSL v.1.0, http://www.globus.org/gram/rsl_spec1.html, 2000*b*.

van der Aalst, W., H. Verbeek und A. Kumar, XRL/Woflan: Verification and extensibility of an XML/Petri-net based language for inter-organizational workflows, in *6th INFORMS Conference on Information Systems and Technology (CIST-2001)*, 2001.

van der Aalst, W. und A. Kumar, XML based schema definition for support of inter-organizational workflow, 2000.

von Laszewski, G., XML schema representation of a simple compute resources, technical report, Mathematics and Computer Science Division at Argonne National Laboratory, computingportals.tacc.utexas.edu/gce_testbed/ globus_computeresource_xml.pdf, 2002.

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.