GWD-R
Distributed Resource Management
Application API (DRMAA) Working Group

Daniel Templeton, Sun Microsystems (editor)
Peter Tröger, University of Potsdam (editor)
Roger Brobst, Cadence Design Systems
Andreas Haas*, Sun Microsystems
Hrabri Rajic*, Intel Americas Inc.
*co-chairs
May, 2005

Distributed Resource Management Application API - IDL Bindings 0.35

Status of This Memo

This memo is a Global Grid Forum Grid Working Draft - `Recommendation` (GWD-R) in process, in general accordance with the provisions of Global Grid Forum Document GFD-C.1, the Global Grid Forum Documents and Recommendations: Process and Requirements, revised April 2002.

Table of Contents

1 Abstract

This document describes the common base for the Distributed Resource Management Application API (DRMAA) language bindings. The document is based on the implementations work of the DRMAA 1.0 GWD-R document.

2 Introduction

This document gives an IDL description for the DRMAA interface. It arises from the results of a collaborative effort to bring the Java™ language binding and .NET language binding into agreement, based on the DRMAA 1.0 specification.
The DRMAA Interface Specification was written originally with a procedural C-language slant. As such, several aspects of the DRMAA interface needed to be altered slightly to better fit with object-oriented languages. Among the aspects that changed are variable and method naming and the error structure.
Although this document can be seen as stand-alone, it still bases on the concepts defined in the DRMAA 1.0 specification. The text refers to the respective chapter of the DRMAA standard whenever it is necessary.

2.1 How to read this document

In this document, the following conventions are used:

- `IDL language elements and definitions` are represented in a fixed-width font.
- *References to IDL language elements and definitions* are represented in italics.

The key words “MUST,” “MUST NOT,” “REQUIRED,” “SHALL,” “SHALL NOT,” “SHOULD,” “SHOULD NOT,” “RECOMMENDED,” “MAY,” and “OPTIONAL” are to be interpreted as described in RFC-2119 [RFC 2119].

The document describes the DRMAA interface semantics with the help of OMG IDL [OMG IDL]. It includes a set of overall rules for the creation of specific language bindings for the given specification. Specific examples are given for the Java language. These examples are not normative.

3 Design Decisions

An effort has been made to choose design patterns that are not unique to a specific language. However, in some cases, various languages disagree over some points. In those cases, the most meritous approach was taken, irrespective of language.
The following text bases on the terminology of OMG IDL. For this reason, all operational semantics are described in terms of interfaces and not of classes. This concept ensures the possibility to map the described operational semantics to a variety of object-oriented, and even procedural, languages. The usage of a class concept depends on the specific language-mapping rules.
This specification assumes that all possible language bindings based on this document can use an introspection concept. Therefore some methods from the DRMAA specification are unnecessary. The drmaa_get_attribute(), drmaa_set_attribute(), drmaa_get_vector_attribute(), drmaa_set_vector_attribute(), and drmaa_get_vector_attribute_names() methods are not needed because introspective languages are able to obtain a list of property names from the *JobTemplate::getAttributeNames()* method and use introspection to locate the appropriate getters and getters. The getters and setters can also be directly accessed.

3.1 Service Provider Interface

The IDL binding approach borrows from the Java platform the notion of a service provider interface. This idea means that a common subset of the API need only be implemented once

for a programming language, whereas each vendor can provide his own service provider implementation. The service provider interface is accomplished by factoring common functionality out into specific interfaces. The service provider then implements these interfaces and uses the specific classes to build his own implementation. The advantage is that the common core functionality need only be implemented once in any given language, and that developers can become familiar with a single package that may hide beneath it several different vendor implementations.

4 General concepts

4.1 IDL language mapping

Language binding documents based on this specification MUST define a mapping between the IDL constructs used in this specification and their specific language constructs. A language binding SHOULD NOT rely itself completely on the OMG language mapping documents available for many programming languages. It must be considered that the OMG mappings bring a huge overhead of irrelevant CORBA-related mapping rules into the specification. Therefore it must be carefully decided whether a binding decision reflects a natural and simple mapping of the intended purpose for the DRMAA interfaces. In most situations it SHOULD be enough to reuse value type mappings only and to define custom mappings for the reference types.

The language binding MUST use the described concept mapping in a consistent manner for the overall specification.

It may be the case that IDL constructs do not map directly to an according language construct. In this case it MUST be ensured that the according construct in the particular language retains the intended semantic of the DRMAA interface definition.

Languages without an explicit notion of enumerations MAY map the IDL enumeration values to constant class members, enabled by the distinct naming of all enumeration values in the specification.

This specification tries to consider the possibility of a Remote Procedure Call scenario in a DRMAA-conformant language mapping. It MUST therefore be ensured that the programming language type for an IDL *valuetype* definition supports the serialization, comparison, cloning and string representation of *valuetype* instances. These capabilities SHOULD be accomplished through whatever mechanism is most natural for the specific programming language. The IDL *valuetype* definitions SHOULD always map to a reference type in the binding specification.

Java binding example:

IDL	*Java language*
`module` definition	`package` keyword
`interface` definition	`public abstract interface` definition
`enum` definition with enumeration members	Enumeration members become Java `int` constants in the surrounding interface definition
`string` type	`java.lang.String`
`long` type	`int`
`long long` type	`long`

`const` type	`public static final`
`boolean` type `[readonly] attribute` type	`boolean` Getter [and setter] methods in JavaBeans™ style, boolean readonly attribute names are prefixed with “get”.
`exception` type	Class definition, derived from `java.lang.Exception`
`raises` clause	`throws` clause
`valuetype` definition	`public class` definition, may additionally implement the *Cloneable*, *Serializable*, and *Compareable* interfaces
`factory` definition	class constructor

The DRMAA IDL definition defines specialized custom types:

```
// unbounded native string list
valuetype StringList sequence<string>;
// dictionary type, for unbounded key-value pair storage
valuetype Dictionary sequence< sequence<string,2> >;
```

The language-binding author SHOULD replace these type definitions directly with semantically equal basic language constructs, if possible. This MAY include the usage of multiple types for one of the above concepts, depending on the context.

Java binding example:

IDL	*Java*
`StringList`	`java.util.List`
`Dictionary`	`java.util.Map`

4.2 The DRMAA Module

The DRMAA IDL binding distinguishes between the application programming interface part (API) and the service provider interface part (SPI):

```
module DRMAA{
  // API part
  ...
  valuetype FileTransferMode{...};
  ...
  // SPI part
  ...
  interface Session{...};
  ...
}
```

Kommentar: This is better, but I still find it to be an awkward distinction. The problem is that our SPI is part of the API. Perhaps we should rephrase out the SPI stuff and explain what vendors have to do to implement it.

The API part contains all definitions and types that are not specific to the underlying DRM system, but specific for the particular programming language. The SPI part defines the subset of functionality that must be implemented by each DRMS vendor separately.
Language binding authors MUST map the IDL module encapsulation to an according package or namespace concept and MAY change the module name according to programming language conventions.

Java binding example:

IDL	*Java*
`module DRMAA`	`package org.ggf.drmaa`

5 Application Programming Interface (API) Section

The API part of the DRMAA module defines the vendor-independent, language-dependent parts of the DRMAA programming interface. It consists of several DRMAA-related data structures and the possible exception types.

```
module DRMAA{
          // API part
          enum JobControlAction {…};
          enum JobProgramState {…};
          enum JobSubmissionState {…};
          valuetype FileTransferMode {…};
          valuetype Version {…};
          exception AuthorizationException {…};
          exception InvalidContactStringException {…};
          exception DefaultContactStringException {…};
          exception NoDefaultContactStringSelectedException {…};
          exception DeniedByDrmException {…};
          exception DrmCommunicationException {…};
          exception DrmsExitException {…};
          exception HoldInconsistentStateException {…};
          exception ReleaseInconsistentStateException {…};
          exception ResumeInconsistentStateException {…};
          exception SuspendInconsistentStateException {…};
          exception DrmsInitException {…};
          exception InvalidArgumentException {…};
          exception InvalidJobException {…};
          exception ConflictingAttributeValuesException {…};
          exception InvalidAttributeFormatException {…};
          exception InvalidAttributeValueException {…};
          exception NoResourceUsageException {…};
          exception ExitTimeoutException {…};
          exception NoActiveSessionException {…};
          exception AlreadyActiveSessionException {…};
          exception TryLaterException {…};
          exception InternalException {…};
          exception OutOfMemoryException {…};
          exception UnsupportedAttributeException {…};
          exception InvalidJobTemplateException {…};
          native PartialTimestamp;
          // SPI part
          ...
};
```

5.1 JobControlAction enumeration

The *JobControlAction* enumeration is used as a input parameter type by the *control()* method in the *Session* interface. The meanings of the enumeration values are specified in the description of the method in section 6.3.8.

```
enum JobControlAction {
          SUSPEND,
          RESUME,
          HOLD,
          RELEASE,
          TERMINATE
};
```

5.2 JobProgramState enumeration

The *JobProgramState* enumeration is used as a input parameter type by the *jobProgramStatus()* method in the *Session* interface. The meanings of the enumeration values are specified in the description of the method in section 6.3.11.

```
enum JobProgramState {
          UNDETERMINED,
          QUEUED_ACTIVE,
          SYSTEM_ON_HOLD,
          USER_ON_HOLD,
          USER_SYSTEM_ON_HOLD,
          RUNNING,
          SYSTEM_SUSPENDED,
          USER_SUSPENDED,
          USER_SYSTEM_SUSPENDED,
          DONE,
          FAILED
};
```

5.3 JobSubmissionState enumeration

The *JobSubmissionState* enumeration is used as the type of the *JobTemplate::jobSubmissionState* interface attribute. In the context of the job template, the enumeration values have the following meaning:

- *HOLD_STATE:* The job may be queued but it is not eligible to run.
- *ACTIVE_STATE:* The job is currently running.

```
enum JobSubmissionState {
          HOLD_STATE,
          ACTIVE_STATE
};
```

5.4 FileTransferMode value type

The *FileTransferMode* value-type is used by the *JobTemplate* interface to indicate the value for the *transferFiles* attribute. The type contains three attributes which determine the streams that will be staged in or out.

```
valuetype FileTransferMode {
          attribute boolean transferInputStream;
          attribute boolean transferOutputStream;
          attribute boolean transferErrorStream;
          factory FileTransferMode();
          factory FileTransferMode(in boolean transferInputStream,
                                   in boolean transferOutputStream,
                                   in boolean transferErrorStream   );
};
```

5.4.1 transferInputStream

This attribute defines whether to transfer input stream files. If this attribute contains true, the transfer*inputStream* attribute of the corresponding job template SHALL be treated as the source from which input files should be copied.

5.4.2 transferOutputStream

This attribute defines whether to transfer output stream files. If this attribute contains true, the transferO*utputStream* attribute of the corresponding job template SHALL be treated as the destination to which output files should be copied.

5.4.3 transferErrorStream

This attribute defines whether to transfer error stream files. If this attribute contains true, the transferE*rrorStream* attribute of the corresponding job template SHALL be treated as the destination to which error files should be copied.

5.5 Version value type

The *Version* value type is a holding structure for the major and minor version numbers of the DRMAA implementation as contained in the *version* attribute of the *Session* interface. The string representation (see section 4.1) of a *Version* instance MUST be of the form "<major>.<minor>".

```
valuetype Version {
          readonly attribute long major;
          readonly attribute long minor;
          factory Version(in long major, in long minor);
};
```

5.5.1 major

This attribute SHALL contain the major version number.

5.5.2 minor

This attribute SHALL contain the minor version number.

5.6 Exceptions

All exceptions in specific bindings MUST contain a possibility to store and read a textual description of the exception cause for the exception instance.
Language bindings MAY decide to derive all exceptions from given environmental exception base class(es). Language bindings SHOULD replace exceptions with a semantically equivalent native runtime environment exception whenever this is appropriate.

```
exception AlreadyActiveSessionException {string message;};
exception AuthorizationException {string message;};
exception ConflictingAttributeValuesException {string message;};
exception DefaultContactStringException {string message;};
exception DeniedByDrmException {string message;};
exception DrmCommunicationException {string message;};
exception DrmsExitException {string message;};
exception DrmsInitException {string message;};
exception ExitTimeoutException {string message;};
exception HoldInconsistentStateException {string message;};
exception InternalException {string message;};
exception InvalidArgumentException {string message;};
exception InvalidAttributeFormatException {string message;};
exception InvalidAttributeValueException {string message;};
exception InvalidContactStringException {string message;};
exception InvalidJobException {string message;};
exception InvalidJobTemplateException {string message;};
exception NoActiveSessionException {string message;};
exception NoDefaultContactStringSelectedException {string message;};
exception NoResourceUsageException {string message;};
exception OutOfMemoryException {string message;};
exception ReleaseInconsistentStateException {string message;};
exception ResumeInconsistentStateException {string message;};
exception SuspendInconsistentStateException {string message;};
exception TryLaterException {string message;};
exception UnsupportedAttributeException {string message;};
```

Language bindings MAY decide to introduce a hierarchical ordering of the DRMAA exceptions through class derivation. In this case it MAY also happen that new exceptions are introduced for behavior aggregation. In this case, those exceptions SHALL be marked as abstract.

If the language supports the distinction between static (‘checked’) and runtime (‘unchecked’) exceptions, all but the following exceptions must be represented as checked exception:

- `InternalException`
- `OutOfMemoryException`
- `UnsupportedAttributeException`
- `InvalidJobTemplateException`
- `AuthorizationException`

5.6.1 AlreadyActiveSessionException

Initialization failed due to existing DRMAA session.

5.6.2 AuthorizationException

The user is not authorized to perform the given operation.

5.6.3 ConflictingAttributeValuesException

The value of this attribute conflicts with one or more previously set properties.

5.6.4 DefaultContactStringException

The DRMAA implementation could not use the default contact string to connect to DRM system.

5.6.5 DeniedByDrmException

The DRM system rejected the job. The job will never be accepted due to DRM configuration or job template settings.

5.6.6 DrmCommunicationException

Could not contact DRM system.

5.6.7 DrmsExitException

A problem was encountered while trying to exit the session.

5.6.8 DrmsInitException

A problem was encountered while trying to initialize the session.

5.6.9 ExitTimeoutException

The wait() or synchronize() method call on the Session interface returned before all selected jobs entered the *DONE* or *FAILED* state.

5.6.10 HoldInconsistentStateException

The job cannot be moved to a *HOLD* state.

5.6.11 InternalException

Unexpected or internal DRMAA error like system call failure, etc.

5.6.12 InvalidArgumentException

A parameter value is fundamentally invalid, such as being of the wrong type or being `null`.

5.6.13 InvalidAttributeFormatException

The value for the job template property is improperly formatted, such as a badly formatted time stamp.

5.6.14 InvalidAttributeValueException

The value for the job template property is invalid.

5.6.15 InvalidContactStringException

The given contact string is not valid.

5.6.16 InvalidJobException

The job specified by the given job id does not exist.

5.6.17 InvalidJobTemplateException

The job template is not valid. It was either created incorrectly, i.e. not via *Session::createJobTemplate(),* or it has already been deleted via *Session::deleteJobTemplate()* method.

5.6.18 NoActiveSessionException

Method call failed because there is no active session.

5.6.19 NoDefaultContactStringSelectedException

No defaults contact string was provided or selected. DRMAA requires that the default contact string is selected when there is more than one default contact string due to multiple DRMAA implementations being present and available.

5.6.20 NoResourceUsageException

This exception is thrown by *Session::wait()* when a job has finished but no resource usage or exit status data could be provided.

5.6.21 OutOfMemoryException

This exception can be throw by any method at any time when the DRMAA implementation has run out of free memory.

5.6.22 ReleaseInconsistentStateException

The job is not in a *HOLD* state, and hence cannot be released.

5.6.23 ResumeInconsistentStateException

The job is not in a suspended state (**_SUSPENDED)*, and hence cannot be resumed.

5.6.24 SuspendInconsistentStateException

The job is not is a state from which it can be suspended.

5.6.25 TryLaterException

The DRMS rejected the operation due to excessive load. A retry attempt may succeed, however.

5.6.26 UnsupportedAttributeException

The given job template attribute is not supported by the current DRMAA implementation.

5.7 The PartialTimestamp format

The *PartialTimestamp* type is used by the *JobTemplate* interface to represent partially specified time stamps, as required by the Distributed Resource Management Application API Specification 1.0. The *PartialTimestamp* SHOULD be an extension of the native language date/time representation if possible and reasonable. For this reason, the following text describes the functional requirements without a specific signature for the type definition. The IDL definition covers this aspect by specifying a native data type.

```
native PartialTimestamp;
```

The *PartialTimestamp* MUST support the following fields: century (>=19), year (0-99), month (1-12), date (1-31), hour (0-23), minute (0-59), second (0-61), zone offset hour (-11 - 12), and zone offset minute (0-59). It MUST support the following essential operations: “get field value”, “set field value”, “get time as native date/time object”, “convert to string” and “parse from string.” If possible, these operations SHOULD leverage structure already present in the native date/time class, even if this leads to a mapping with multiple classes or interfaces. The two field operations MAY be represented as attributes.
The “get field value” operation MUST return the current value for the given field. The “set field value” operation MUST set the current value for the given field. The “get time as native date/time object” operation MUST resolve the partial time to a specific time that is the soonest possible time that is not in the past, and SHOULD return that specific time as a native date/time representation. The “convert to string” operation MUST return the partial time represented by the *PartialTimestamp* as a String which adheres to the following format:
`[[[[CC]YY/]MM/]DD] hh:mm[:ss] [{-|+}UU:uu]`, where:

- CC is the first two digits of the year [19,]
- YY is the last two digits of the year [0,99]
- MM is the two digits of the month [01,12]
- DD is the two-digit day of the month [01,31]
- hh is the two-digit hour of the day [00,23]
- mm is the two-digit minute of the day [00,59]
- ss is the two-digit second of the minute [00,61]
- UU is the two-digit hours since (before) UTC [-11,12]
- uu is the two-digit minutes since (before) UTC [0,59]

In order for this operation to be performed, the *PartialTimestamp* must have no unset field of a lower order than the highest order set field, with the exception of the second and zone offset fields. For example, if the year is set, the month, date, hour, and minute must also be set for this operation to be performed. Failure to meet this criterion MUST result in an *InvalidArgumentException* being thrown, or the corresponding error code being returned in languages which do not support exceptions. The “parse from string” operation MUST parse a string in the above format to generate a *PartialTimestamp* as the return value. If the string is not in the above format, an *InvalidArgumentException* or an appropriate language-dependent exception MUST be thrown or the corresponding error code MUST be returned in languages that do not support exceptions.
The resolving of partial time information must be performed according to the following rules:

- If the optional UTC-offset is not specified, the offset associated with the local timezone SHALL be used.
- If the second is not specified, then it should be set to zero.

- If the day is not specified, the current day SHALL be used unless the specified hour, minute and second has already elapsed, in which case the next day SHALL be used.
- If the month is not specified, the current month SHALL be used unless the specified day, hour, minute and second has already elapsed, in which case the next month SHALL be used.
- If the year is not specified, the current year SHALL be used unless the specified month, day, hour, minute and second has already elapsed, in which case the next year SHALL be used.
- If the century is not specified, the current century SHALL be used unless the specified year, month, day, hour, minute and second has already elapsed, in which case the next century SHALL be used.

The *PartialTimestamp* MAY also support the following four operations: "get field modifier," "set field modifier," "add to field," and "roll field." If possible, these operations SHOULD leverage structure already present in the native language date/time representation. The "get field modifier" operation MUST return any additional modifiers set for the given field. An additional modifier is added to the field's value after it has been resolved to a specific time. The "set field modifier" operation MUST set the additional modifiers for the given field. The "add to field" operation MUST add a given value to the given field. If supported by the native date/time representation, this operation SHOULD attempt to resolve out of range field values that may result from the operation. For example, adding "1" to the date of a *PartialTimestamp* instance which is set to January 31st SHOULD result in the *PartialTimestamp* being set to February 1st. If this operation is supported, the "get field modifier" and "set field modifier" operations MUST also be supported. The "roll field" operation is similar to the "add to field" operation, except that the operation cannot modify a field of a higher order than the given field. Such modifications are simply lost. For example, adding "1" to the date of a *PartialTimestamp* which is set to January 31st SHOULD result in the *PartialTimestamp* being set to January 1st.
The *PartialTimestamp* MUST also support a notion of unset fields. A special value is assigned to all fields which have not been explicitly set. This special value MUST be of the same type as the date/time properties and MAY be the maximum value for that data type.
Language bindings are free to define convenience functions in addition to the functionalities described here.

6 Service Provider Interface (SPI) Section

The SPI part of the DRMAA module consists of several interfaces. The *Session* interface represents the majority of the functionality defined by the DRMAA specification. It utilizes all the data structures defined in the API and SPI section.

```
module DRMAA{
 // API part
 ...
 // SPI part
 interface JobInfo {…};
 interface JobTemplate {…};
 interface Session{
     void init(in string contactString);
     void exit();
     JobTemplate createJobTemplate();
     void deleteJobTemplate(in JobTemplate jobTemplate);
     string runJob(in JobTemplate jobTemplate);
     StringList runBulkJobs(
          in JobTemplate jobTemplate,
          in long beginIndex,
          in long endIndex,
```

```
        in long step);
    void control(in string jobName,in JobControlAction operation);
    void synchronize(
        in StringList jobList,
        in long long timeout,
        in boolean dispose);
    JobInfo wait(
        in string jobName,
        in long long timeout);
    JobProgramState jobProgramStatus(in string jobName);
    readonly attribute string contact;
    readonly attribute Version version;
    readonly attribute string drmsInfo;
    readonly attribute string drmaaImplementation;
 };
};
```

6.1 JobInfo interface

The information regarding a job's execution is encapsulated in instances that fulfil the *JobInfo* interface. With the help of the *JobInfo* attributes, an application can discover information about the resource usage and exit status of a job. The structure of the *JobInfo* interface is as follows:

```
interface JobInfo {
 readonly attribute string jobId;
 readonly attribute Dictionary resourceUsage;
 readonly attribute boolean exited;
 readonly attribute long exitStatus;
 readonly attribute boolean signaled;
 readonly attribute string terminatingSignal;
 readonly attribute boolean coreDump;
 readonly attribute boolean aborted;
 readonly attribute string reason;
};
```

Kommentar: I thought we were going to go back to the DRMAA spec naming, i.e. ifExited(), coreDump(), etc.

Kommentar: If you perform the name mapping to Java as described in the first part, you come out with more or less the same names for the JobInfo members as in the original spec. Therefore I don't see the problem.

The following sections explain the meanings of the *JobInfo* member attributes.

6.1.1 jobId

The identifier of the completed job.

6.1.2 resourceUsage

The completed job's resource usage data.

6.1.3 exited

This attribute SHALL contain *true* if the job terminated normally. *False* MAY also indicate that although the job has terminated normally, an exit status is not available, or that it is not known whether the job terminated normally. In both cases the *exitStatus* attribute SHALL NOT contain exit status information. *True* indicates more detailed diagnosis can be retrieved from the *exitStatus* attribute.

6.1.4 exitStatus

If *exited* is *true*, this attribute contains the operating system exit code of the job.

Kommentar: This attribute should throw an exception if exited is false, but which one ? Maybe UnsupportedAttributeException, or something new ?

6.1.5 signaled

This attribute SHALL contain *true* if the job terminated due to the receipt of a signal. *False* MAY also indicate that although the job has terminated due to the receipt of a signal, the signal is not available, or that it is not known whether the job terminated due to the receipt of a signal. In both cases *terminatingSignal* SHALL not provide signal information.

Kommentar: This is still a bit of ugliness. There has to be a better way to handle interrogating the info object. What about some kind of enumeration instead of the series of function calls? We may need to ask the group why they decided to do it that way in the first place.

6.1.6 terminatingSignal

If *signaled* is *true*, this attribute SHALL contain a representation of the signal that caused the termination of the job. For signals declared by POSIX, the symbolic names SHALL be returned (e.g., SIGABRT, SIGALRM). For signals not declared by POSIX, a DRM dependent string SHALL be returned.

Kommentar: This attribute should throw an exception if signaled is false, but which one ? (see above)

6.1.7 coreDump

If *signaled* is *true*, this attribute SHALL contain true if a core image of the terminated job was created.

6.1.8 aborted

Kommentar: This attribute should throw an exception if signaled is false, but which one ? (see above)

This attribute SHALL contain *true* if the job ended before entering the running state.

6.1.9 reason

If *aborted* is *true*, this attribute SHALL contain a string representation of the reason why the job aborted. This string representation SHALL be implementation dependent.

Kommentar: This attribute should throw an exception if aborted is false, but which one ? (see above)

6.2 JobTemplate interface

In order to define the attributes associated with a job, a DRMAA application uses the *JobTemplate* interface. Instances of such templates are created via the active *Session* implementation. A DRMAA application gets a *JobTemplate* from the active *Session* instance, specifies in the template any required job parameters, and the passes the template back to the session when requesting that a job be executed. When finished, the DRMAA application should call the *Session::deleteJobTemplate()* method to allow the underlying implementation to free any resources bound to the *JobTemplate* instance. The structure of the *JobTemplate* interface is as follows:

```
interface JobTemplate{
 const string HOME_DIRECTORY = "$drmaa_hd_ph$";
 const string WORKING_DIRECTORY = "$drmaa_wd_ph$";
 const string PARAMETRIC_INDEX = "$drmaa_incr_ph$";
 attribute string remoteCommand;
 attribute StringList args;
 attribute JobSubmissionState jobSubmissionState;
 attribute Dictionary jobEnvironment;
 attribute string workingDirectory;
        attribute string jobCategory;
 attribute string nativeSpecification;
 attribute StringList email;
 attribute boolean blockEmail;
 attribute PartialTimestamp startTime;
        attribute string jobName;
 attribute string inputPath;
```

```
 attribute string outputPath;
 attribute string errorPath;
 attribute boolean joinFiles;
 attribute FileTransferMode transferFiles;
        attribute PartialTimestamp deadlineTime;
 attribute long long hardWallclockTimeLimit;
 attribute long long softWallClockTimeLimit;
 attribute long long hardRunDurationLimit;
 attribute long long softRunDurationLimit;
 StringList getAttributeNames()
          raises (  DrmCommunicationException,
                    AuthorizationException,
                    NoActiveSessionException,
                    OutOfMemoryException,
                    InvalidArgumentException,
                    InternalException);
};
```

The *JobTemplate* implementation MUST support two types of exceptions for the setter operations in case there is such a concept in the programming language:

- *InvalidAttributeValueException* – The value is invalid for the job template property, e.g. a *startTime* that is in the past.
- *ConflictingAttributeValuesException* – the attribute value conflicts with a previously set attribute value.

In most cases, a DRMAA implementation will require that job templates be created through the *Session::createJobTemplate()* method. In those cases, passing a template created other than via this method to the *Session::deleteJobTemplate()*, *Session::runJob()*, or *Session::runBulkJobs()* methods MUST result in an *InvalidJobTemplateException* being thrown or a corresponding error code being returned if exceptions are not supported.

A *JobTemplate* instance must be convertible to a String for printing. This should be accomplished through whatever mechanism is most natural for the implementation language. The resulting String MUST contain the values of all set properties.

The access to attribute values MUST operate in a pass-by-value mode. An according language binding must ensure that this behavior is always fulfilled.

In the job template there is a distinction between mandatory and optional attributes. A language binding implementation MUST provide implementations for all DRMAA attributes, both required and optional. The setter and getter implementations for optional attributes MUST throw *UnsupportedAttributeException* in languages which support exceptions. In languages which do not support exceptions, the optional attribute setters and getters MUST return some form of error. The service provider implementation SHOULD then override the setters and getters for supported optional attributes with methods that operate normally.

The SPI implementation is also allowed to add implementation-specific attributes. The *JobTemplate::getAttributeNames()* method SHALL return the names of all job template attributes supported by the service provider implementation, including required, optional, and implementation specific attributes. In order to get the values for supported attributes, such as in a property sheet, one should use introspection to call the appropriate setter and getter for each attribute.

Kommentar: Languages without introspection need setAttribute() and getAttribute(). Please note that there is an agreed extension of the possible error codes of this functions (see GridForge issues #1177, #1178 and #1180)

Kommentar: Printing & Cloning for a non-value type may be a problem

6.2.1 Constants

The *JobTemplate* interface defines a set of constants which are used in the context of some of the attributes:

```
const string HOME_DIRECTORY = "$drmaa_hd_ph$";
const string WORKING_DIRECTORY = "$drmaa_wd_ph$";
const string PARAMETRIC_INDEX = "$drmaa_incr_ph$";
```

The *HOME_DIRECTORY* constant is a place holder used to represent the user's home directory when building paths for the *workingDirectory*, *inputPath*, *outputPath*, and *errorPath* attributes.

The *WORKING_DIRECTORY* constant is a place holder used to represent the current working directory when building paths for the *inputPath*, *outputPath*, and *errorPath* attributes.

The *PARAMETRIC_INDEX* constant is a place holder used to represent the id of the current parametric job subtask when building paths for the *workingDirectory*, *inputPath*, *outputPath*, and *errorPath* attributes.

Kommentar: Depending on GridForge tracker item #805 we might want to support some other placeholders.

6.2.2 remoteCommand

The command that should be executed on the remote host. In case this parameter contains path information, it MUST be seen as relative to the execution host file system and is therefore evaluated there. The attribute value SHOULD NOT relate to binary file management or file staging activities.

6.2.3 args

The list of command-line arguments for the job to be executed.

6.2.4 jobSubmissionState

Defines the state of the job at submission time. For more information see section 5.3.

6.2.5 jobEnvironment

The environment values that define the remote environment. The values MUST override the remote environment values if there is a collision. If this is not possible, the behaviour is implementation dependent.

6.2.6 workingDirectory

This attribute specifies the directory where the job is executed. If the attribute is not set, the behaviour is is implementation dependent. The attribute value MUST be evaluated relative to the execution host file system. The attribute value MAY contain the *HOME_DIRECTORY* or *PARAMETRIC_INDEX* constant values as placeholder. A *HOME_DIRECTORY* placeholder at the begin denotes the remaining portion of the attribute value as a relative directory name resolved relative to the job users home directory at the execution host. The *PARAMETRIC_INDEX* placeholder MAY be used at any position within the attribute value in case of parametric job templates and SHALL be substituted by the underlying DRM system with the parametric jobs' index.
The *workingDirectory* MUST be specified in a syntax that is common at the host where the job is executed.

If the attribute is set and no placeholder is used, an absolute directory specification is expected. If the attribute is set and the directory does not exist, the job enters the state *JobProgramState.FAILED*.

6.2.7 jobCategory

An implementation-defined string specifying how to resolve site-specific resources and/or policies. Site administrators MAY create a job category suitable for an application to be dispatched by the DRMS; the associated category name SHALL be specified as a job submission attribute. The DRMAA implementation MAY then use the category name to manage site-specific resource and functional requirements of jobs in the category. Such requirements need to be configurable by the site operating a DRMS and deploying an application on top of it. More information can be found in section 2.4.1 of the DRMAA 1.0 specification document.

6.2.8 nativeSpecification

An implementation-defined string that is passed by the end user to DRMAA to specify site-specific resources and/or policies.
As far as the DRMAA interface specification is concerned, the native specification is an implementation-defined string and is interpreted by each DRMAA library. One MAY use job categories and native specification with the same job submission for policy specification. In this case, the DRMAA library is assumed to be capable of joining the outcome of the two policy sources in a reasonable way.
Native specification MAY be used without the requirement to maintain job categories, and submit options MAY be specified directly.
More information can be found in section 2.4.2 of the DRMAA 1.0 specification document.

6.2.9 email

A list of email addresses that is used to report the job completion and status.

6.2.10 blockEmail

This Boolean parameter decides whether the sending of email is blocked by default or not, regardless of the DRMS setting.

6.2.11 startTime

This attribute specifies the earliest time when the job MAY be eligible to be run.

6.2.12 jobName

A job name SHALL comprise alphanumeric and _ characters.The DRMAA implementation MAY truncate any client-provided job name to an implementation-defined length that is at least 31 characters.

6.2.13 inputPath

Specifies the job standard input as path to a file. Unless set elsewhere, if not explicitly set in the job template, the job is started with an empty input stream. If set, specifies the network path of the jobs input stream file of the form

```
[hostname]:file_path
```

When the *transferFiles* job template attribute is supported and has a value where the *FileTransferMode::inputStream* attribute set to *true*, the input file SHOULD be fetched by the underlying DRM system from the specified host, or from the submit host if no hostname was specified.
When the *transferFiles* job template attribute is not supported or it's values member *FileTransferMode::inputStream* is set to *false*, then the input file is always expected at the host where the job is executed, irrespective of a possibly hostname specified.
The *PARAMETRIC_INDEX* placeholder can be used at any position for parametric job templates and SHALL be substituted by the underlying DRM system with the parametric jobs' index.
A *HOME_DIRECTORY* placeholder at the begin of the attribute value denotes the remaining portion as a relative file specification resolved relative to the job users home directory at the host where the file is located.
A *WORKING_DIRECTORY* placeholder at the begin of the attribute value denotes the remaining portion as a relative file specification resolved relative to the jobs working directory at the host where the file is located.
The *inputPath* MUST be specified in a syntax that is common at the host where the file is located.
If set, and the file can't be read, the job enters the state *JobProgramState.FAILED*.

6.2.14 outputPath

Specifies how to direct the jobs' standard output to a file. If not explicitly set in the job template, the whereabouts of the jobs output stream is not defined. If set, specifies the network path of the jobs output stream file of the form

```
[hostname]:file_path
```

When the *transferFiles* job template attribute is supported and its value's member *FileTransferMode::outputStream* attribute is set to *true*, the output file SHALL be transferred by the underlying DRM system to the specified host or to the submit host if no hostname is specified.
When the *transferFiles* job template attribute is not supported or the *FileTransferMode::outputStream* attribute is set to *false,* then the output file is always kept at the host where the job is executed irrespectively of a possibly hostname specified.
The *PARAMETRIC_INDEX* placeholder can be used at any position with parametric job templates and SHALL be substituted by the underlying DRM system with the parametric jobs' index.
A *HOME_DIRECTORY* placeholder at the begin denotes the remaining portion as a relative file specification resolved relative to the job users home directory at the host where the file is located.
A *WORKING_DIRECTORY* placeholder at the begin denotes the remaining portion as a relative file specification resolved relative to the jobs working directory at the host where the file is located.
The *outputPath* MUST be specified in a syntax that is common at the host where the file is located. If set and the file can't be written before execution the job enters the state *JobProgramState.FAILED*.

6.2.15 errorPath

Specifies how to direct the jobs' standard error to a file.
If not explicitly set in the job template, the whereabouts of the jobs error stream is not defined. If set, specifies the network path of the jobs error stream file of the form

```
[hostname]:file_path
```

When the transferFiles job template attribute is supported and in it's value the *FileTransferMode::errorStream* attribute is set, the output file SHALL be transferred by the underlying DRM system to the specified host or to the submit host if no hostname is specified. When the *FileTransferMode::errorStream* attribute is not supported or it's value does have the *FileTransferMode::errorStream* set to *false*, the error file is always kept at the host where the job is executed irrespectively of a possibly hostname specified.
The *PARAMETRIC_INDEX* placeholder can be used at any position for parametric job templates and SHALL be substituted by the underlying DRM system with the parametric jobs' index.
A *HOME_DIRECTORY* placeholder at the begin denotes the remaining portion as a relative file specification, resolved relative to the job users home directory at the host where the file is located.
A *WORKING_DIRECTORY* placeholder at the begin denotes the remaining portion as a relative file specification resolved relative to the jobs working directory at the host where the file is located.
The *errorPath* MUST be specified in a syntax that is common at the host where the file is located.
If set and the file can't be written before execution the job enters the state *JobProgramState.FAILED*.

6.2.16 joinFiles

Specifies if the error stream should be intermixed with the output stream. If not explicitly set in the job template the attribute defaults to *false*. If *true* is specified the underlying DRM system SHALL ignore the value of the *errorPath* attribute and intermix the standard error stream with the
standard output stream as specified with *outputPath*.

6.2.17 transferFiles

Specifies how to transfer files between hosts.
If not explicitly set in the job template, all members of the *FileTransferMode* type are non-set. This attribute works in conjunction with the *inputPath*, *outputPath* and *errorPath* attributes.

This attribute is optional. In case an implementation MUST throw an *UnsupportedAttributeException* if this attribute is not supported.

6.2.18 deadlineTime

Specifies a deadline after which the DRMS will terminate a job.

This attribute is optional. In case an implementation MUST throw an *UnsupportedAttributeException* if this attribute is not supported.

6.2.19 hardWallclockTimeLimit

This attribute specifies when the job's wall clock time limit has been exceeded. The implementation SHALL terminate a job that has exceeded its wall clock time limit. Suspended time SHALL also be accumulated here. The value MUST be given in seconds.

This attribute is optional. In case an implementation MUST throw an *UnsupportedAttributeException* if this attribute is not supported.

6.2.20 softWallClockTimeLimit

This attribute specifies an estimate as to how long the job will need wall clock time to complete. Note that the suspended time is also accumulated here. This attribute is intended to assist the scheduler. If the time specified in insufficient, the implementation MAY impose a scheduling penalty. The value MUST be given in seconds.

This attribute is optional. In case an implementation MUST throw an *UnsupportedAttributeException* if this attribute is not supported.

6.2.21 hardRunDurationLimit

This attribute specifies how long the job MAY be in a running state before its limit has been exceeded, and therefore is terminated by the DRMS. The value MUST be given in seconds.

This attribute is optional. In case an implementation MUST throw an *UnsupportedAttributeException* if this attribute is not supported.

6.2.22 softRunDurationLimit

This attribute specifies an estimate as to how long the job will need to remain in a running state to complete. This attribute is intended to assist the scheduler. If the time specified in insufficient, the implementation MAY impose a scheduling penalty.

This attribute is optional. In case an implementation MUST throw an *UnsupportedAttributeException* if this attribute is not supported.

6.2.23 getAttributeNames

This method SHALL return the list of supported attribute names. This list includes supported DRMAA reserved attribute names (both required and optional) and implementation-specific attribute names.

```
 StringList getAttributeNames()
         raises (  DrmCommunicationException,
                   AuthorizationException,
                   NoActiveSessionException,
                   OutOfMemoryException,
                   InternalException);
};
```

Exceptions

- DrmCommunicationException – the DRMS could not be contacted for this request.
- AuthorizationException – the user does not have permission to perform this action.
- NoActiveSessionException – the session has not been initialized or exit() has already been called
- OutOfMemoryException – the DRMAA implementation does not have enough free memory to perform the operation.
- InternalException – an error has occurred in the DRMAA implementation.

6.3 Session interface

The following chapter explains the set of constants, methods and attributes defined in the Session interface.

6.3.1 Constants

The Session interface defines a set of constant values, which are used in the context of several interface functions.

```
const long long TIMEOUT_WAIT_FOREVER = -1;
const long long TIMEOUT_NO_WAIT = 0;
const string JOB_IDS_SESSION_ANY = "DRMAA_JOB_IDS_SESSION_ANY";
const string JOB_IDS_SESSION_ALL = "DRMAA_JOB_IDS_SESSION_ALL";
```

The *TIMEOUT_WAIT_FOREVER* constant is used with the *wait()* and *synchronize()* methods to indicate that the methods should not return until the given job or jobs have entered the *DONE* or *FAILED* state.

The *TIMEOUT_NO_WAIT* constant is used with the *wait()* and *synchronize()* methods to indicate that the methods should return immediately if the given job or jobs have not yet entered the *DONE* or *FAILED* state.

The *JOB_IDS_SESSION_ANY* constant is used with the *wait()* method to indicate that the method may operate on any job currently in the *RUNNING* state in the session.

The *JOB_IDS_SESSION_ALL* constant is used with the *control()* and *synchronize()* methods to indicate that the methods should operate on all jobs currently in the *RUNNING* state in the session.

6.3.2 init

The *init()* method is used to initialize a DRMAA session for use. The *contactString* parameter is an implementation-dependent string that may be used to specify which DRM system to use. This method must be called before any other DRMAA calls, except for the getter functions of the *contact*, *drmsInfo*, and *drmaaImplementation* attributes defined in the Session interface. If contact is `null`, the default DRM system is used, provided there is only one DRMS available. If contact is `null`, and more than one DRMAA implementation is available, *init()* SHALL throw a *NoDefaultContactStringSelectedException* or return a corresponding error code if exceptions aren't supported. *init()* SHOULD be called only once, by only one of the threads. The main thread is recommended. A call to *init()* by another thread or additional calls to *init()* by the same thread with throw a *AlreadyActiveSessionException* or return a corresponding error code if exceptions are not supported.

```
void init(in string contactString)
          raises (  DrmsInitException,
                    InvalidContactStringException,
                    AlreadyActiveSessionException,
                    DefaultContactStringException,
                    NoDefaultContactStringSelectedException,
                    OutOfMemoryException,
                    DrmCommunicationException,
                    AuthorizationException,
                    InvalidArgumentException,
                    InternalException);
```

Parameters

`contact` - implementation-dependent string that may be used to specify which DRM system to use. If `null`, will select the default DRM system if there is only one DRMS available.

Exceptions

- DrmsInitException – failed while initializing the session.
- InvalidContactStringException – the *contact* parameter is invalid.
- AlreadyActiveSessionException – the session has already been initialized.
- DefaultContactStringException – the *contact* parameter is `null` and the default contact string could not be used to connect to the DRMS.
- NoDefaultContactStringSelectedException – the *contact* parameter is `null` and more than one DRMS is available.
- OutOfMemoryException – the DRMAA implementation does not have enough free memory to perform the operation.
- DrmCommunicationException – the DRMS could not be contacted for this request.
- AuthorizationException – the user does not have permission to perform this action.
- InvalidArgumentException – an argument value is invalid.
- InternalException – an error has occurred in the DRMAA implementation.

6.3.3 exit

The *exit()* is used to disengage from the DRM and allow the DRMAA implementation to perform any necessary internal cleanup. This method ends the current DRMAA session but doesn't affect any jobs (e.g., queued and running jobs remain queued and running). *exit()* should be called only once, by only one of the threads. Additional calls to *exit()* beyond the first SHALL throw a *NoActiveSessionException* or return a corresponding error code if exceptions aren't supported.

```
void exit()
          raises (  DrmsExitException,
                    NoActiveSessionException,
                    DrmCommunicationException,
                    AuthorizationException,
                    OutOfMemoryException,
                    InternalException);
```

Exceptions

- DrmsExitException – failed while exiting the session.
- NoActiveSessionException – the session has not been initialized or *exit()* has already been called
- DrmCommunicationException – the DRMS could not be contacted for this request.
- AuthorizationException – the user does not have permission to perform this action.
- OutOfMemoryException – the DRMAA implementation does not have enough free memory to perform the operation.
- InternalException – an error has occurred in the DRMAA implementation.

6.3.4 createJobTemplate

The *createJobTemplate()* method SHALL return a new job template. The job template is used to set the defining characteristics for jobs to be submitted. Once the job template has been created, it should also be deleted (via *deleteJobTemplate()*) when no longer needed. Failure to do so may result in a memory leak.

```
JobTemplate createJobTemplate()
          raises (  DrmCommunicationException,
                    NoActiveSessionException,
                    OutOfMemoryException,
                    AuthorizationException,
                    InternalException);
```

Returns

The *createJobTemplate()* method SHALL return a blank *JobTemplate* instance.

Exceptions

- DrmCommunicationException – unable to communicate with the DRMS
- NoActiveSessionException – the session has not been initialized or *exit()* has already been called
- OutOfMemoryException – the DRMAA implementation does not have enough free memory to perform the operation.
- AuthorizationException – the user does not have permission to perform this action.
- InternalException – an error has occurred in the DRMAA implementation.

6.3.5 deleteJobTemplate

The *deleteJobTemplate()* method is used to deallocate a job template, and SHALL perform all necessary steps required to free all memory associated with this job template.
In languages where memory is not freed explicitly, e.g. languages that use garbage collectors, this method SHALL perform all necessary steps required to prepare this job template to be freed. In languages where finalizers are supported, the implementation of this method MAY be empty.
This method SHALL have no effect on running jobs. This method MUST only work on *JobTemplate* instances that were created with the *createJobTemplate()* method and have not previously been deleted with the *deleteJobTemplate()* method and MUST otherwise throw an *InvalidJobTemplateException.*

```
void deleteJobTemplate(in JobTemplate jobTemplate)

          raises (  DrmCommunicationException,
                    NoActiveSessionException,
                    OutOfMemoryException,
                    AuthorizationException,
                    InvalidArgumentException,
                    InvalidJobTemplateException,
                    InternalException);
```

Parameters

`jobTemplate` - the *JobTemplate* instance to delete.

Exceptions

- DrmCommunicationException – unable to communicate with the DRMS.
- NoActiveSessionException – the session has not been initialized or *exit()* has already been called.
- OutOfMemoryException – the DRMAA implementation does not have enough free memory to perform the operation.
- AuthorizationException – the user does not have permission to perform this action.

- InvalidArgumentException – the argument value is invalid.
- InvalidJobTemplateException – the given job template was not created with *createJobTemplate()* or has already been deleted .
- InternalException – an error has occurred in the DRMAA implementation.

6.3.6 runJob

The *runJob()* method SHALL submit a job with attributes defined in the job template given as a parameter. The returned job identifier SHOULD be a String identical to that returned from the underlying DRM system. This method MUST only work on *JobTemplate* instances that were created with the *createJobTemplate()* method and have not previously been deleted with the *deleteJobTemplate()* method and MUST otherwise throw an *InvalidJobTemplateException.*

```
string runJob(in JobTemplate jobTemplate)
         raises (  TryLaterException,
                   DeniedByDrmException,
                   DrmCommunicationException,
                   AuthorizationException,
                   InvalidJobTemplateException,
                   NoActiveSessionException,
                   OutOfMemoryException,
                   InvalidArgumentException,
                   InternalException);
```

Parameters

`jobTemplate` - the job template to be used to create the job.

Returns

The *runJob()* method SHOULD return a job identifier string identical to that returned from the underlying DRM system.

Exceptions

- TryLaterException – the request could not be processed due to excessive system load.
- DeniedByDrmException – the DRMS rejected the job. The job will never be accepted due to job template or DRMS configuration settings.
- DrmCommunicationException – unable to communicate with the DRMS.
- InvalidJobTemplateException – the given job template was not created with *createJobTemplate()* or has already been deleted.
- AuthorizationException – the user does not have permission to submit jobs.
- NoActiveSessionException – the session has not been initialized or *exit()* has already been called.
- OutOfMemoryException – the DRMAA implementation does not have enough free memory to perform the operation.
- InvalidArgumentException – the argument value is invalid.
- InternalException – an error has occurred in the DRMAA implementation.

6.3.7 runBulkJobs

The *runBulkJobs()* method SHALL submit a set of parametric jobs, dependent on the implied loop index, each with attributes defined in the given job template. Each job in the set is identical except for it's index. The first parametric job has an index equal to *beginIndex*. The next job has an index equal to *beginIndex* + *step*, and so on. The last job has an index equal to

beginIndex + **n** * *step*, where **n** is equal to (*endIndex* – *beginIndex*) / *step*. Note that the value of the last job's index may not be equal to *endIndex* if the difference between *beginIndex* and *endIndex* is not evenly divisble by *step*. The smallest valid value for *beginIndex* is 1. The largest valid value for *endIndex* is language dependent. The *beginIndex* value must be less than or equal to the *endIndex* value, and only positive index numbers are allowed. The index number can be determined by the job in an implementation specific fashion. The returned job identifiers SHOULD be Strings identical to those returned from the underlying DRM system.

The *JobTemplate* interface defines a *PARAMETRIC_INDEX* placeholder for use in specifying paths. This placeholder is used to represent the individual identifiers of the tasks submitted through this method.

This method MUST only work on JobTemplate instances that were created with the *createJobTemplate()* method and have not previously been deleted with the *deleteJobTemplate()* method and MUST otherwise throw an *InvalidJobTemplateException.*

```
StringList runBulkJobs(      in JobTemplate jobTemplate,
                             in long beginIndex,
                             in long endIndex,
                             in long step)
          raises (  TryLaterException,
                    DeniedByDrmException,
                    DrmCommunicationException,
                    AuthorizationException,
                    InvalidJobTemplateException,
                    NoActiveSessionException,
                    OutOfMemoryException,
                    InvalidArgumentException,
                    InternalException);
```

Parameters

`jobTemplate` - the job template to be used to create the job.
`beginIndex` - the starting value for the loop index.
`endIndex` - the terminating value for the loop index.
`step` - the value by which to increment the loop index each iteration.

Returns

The *runBulkJobs()* method SHOULD return a list of job identifier Strings identical to that returned by the underlying DRM system

Exceptions

- TryLaterException – the request could not be processed due to excessive system load.
- DeniedByDrmException – the DRMS rejected the job. The job will never be accepted due to job template or DRMS configuration settings.
- DrmCommunicationException – unable to communicate with the DRMS.
- InvalidJobTemplateException – the given job template was not created with *createJobTemplate()* or has already been deleted.
- AuthorizationException – the user does not have permission to submit jobs.
- NoActiveSessionException – the session has not been initialized or *exit()* has already been called.
- OutOfMemoryException – the DRMAA implementation does not have enough free memory to perform the operation.
- InvalidArgumentException – an argument value is invalid.
- InternalException – an error has occurred in the DRMAA implementation.

6.3.8 control

The *control()* method SHALL to hold, release, suspend, resume, or kill the job identified by *jobName* respective to the *operation* parameter. If this parameter is equal to *JOB_IDS_SESSION_ALL*, then this method SHALL act on all jobs submitted during this DRMAA session up to the moment *control()* is called.

In case that a call with *JOB_IDS_SESSION_ALL* fails for a partial set of the jobs in the session, the implementation SHALL throw an *InternalException*. The error text of the exception should explain the problem in detail and may give an idea of the current status of the session.

To avoid thread races in multithreaded applications, the DRMAA implementation user should explicitly synchronize this call with any other job submission calls or control calls that may change the number of remote jobs.

The legal values for *operation* and their meanings SHALL be:

- JobControlAction::SUSPEND: stop the job,
- JobControlAction::RESUME: (re)start the job,
- JobControlAction::HOLD: put the job on-hold,
- JobControlAction::RELEASE: release the hold on the job, and
- JobControlAction::TERMINATE: kill the job.

This method SHALL return once the action has been acknowledged by the DRM system, but MAY return before the action has been completed.

Some DRMAA implementations MAY allow this method to be used to control jobs submitted external to the DRMAA session, such as jobs submitted by other DRMAA sessions in other DRMAA implementations or jobs submitted via native utilities.

```
void control(      in string jobName,
                   in JobControlAction operation)
          raises (  DrmCommunicationException,
                    AuthorizationException,
                    ResumeInconsistentStateException,
                    SuspendInconsistentStateException,
                    HoldInconsistentStateException,
                    ReleaseInconsistentStateException,
                    InvalidJobException,
                    NoActiveSessionException,
                    OutOfMemoryException,
                    InvalidArgumentException,
                    InternalException);
```

Parameters

`jobName` - The String id of the job to control.
`operation` - the control action to be taken.

Exceptions

- DrmCommunicationException – unable to communicate with the DRMS.
- AuthorizationException – the user does not have permission to modify jobs.
- ResumeInconsistentStateException – the job is not in a state from which is can be resumed.
- SuspendInconsistentStateException – the job is not in a state from which is can be suspended.
- HoldInconsistentStateException – the job is not in a state from which is can be held.

- ReleaseInconsistentStateException – the job is not in a state from which is can be released.
- InvalidJobException – the job id does not represent a valid job.
- NoActiveSessionException – the session has not been initialized or *exit()* has already been called.
- OutOfMemoryException – the DRMAA implementation does not have enough free memory to perform the operation.
- InvalidArgumentException – an argument value is invalid.
- InternalException – an error has occurred in the DRMAA implementation.

6.3.9 synchronize

This method SHALL wait until all jobs specified by *jobList* have finished execution. If *jobList* contains only *JOB_IDS_SESSION_ALL*, then this method waits for all jobs submitted during this DRMAA session up to the moment *synchronize()* is called.

In case that a call with *JOB_IDS_SESSION_ALL* fails for a partial set of the jobs in the session, the implementation SHALL throw an *InternalException*. The error text of the exception should explain the problem in detail and may give an idea of the current status of the session.

To avoid thread race conditions in multithreaded applications, the DRMAA implementation user should explicitly synchronize this call with any other job submission or control calls that may change the number of remote jobs.

To prevent blocking indefinitely in this call, the caller may use a timeout specifying after how many seconds to block in this call. The constant value *TIMEOUT_WAIT_FOREVER* may be specified to wait indefinitely for a result. The constant value *TIMEOUT_NO_WAIT* may be specified to return immediately if no result is available. If the call exits before the timeout has elapsed, all the jobs have been waited on or there was an interrupt. If the invocation exits on timeout, an *ExitTimeoutException* SHALL be thrown or a corresponding error code returned if exceptions aren't supported. The caller should check system time before and after this call in order to be sure of how much time has passed.

The *dispose* parameter specifies how to treat the reaping of the remote job's internal data record, which includes a record of the job's consumption of system resources during its execution and other statistical information. If set to `true`, the DRM SHALL dispose of the job's data record at the end of the *synchronize()* call. If set to `false`, the data record SHALL be left for future access via the *wait()* method.

```
void synchronize(  in StringList jobList,
                   in long long timeout,
                   in boolean dispose)
          raises (  DrmCommunicationException,
                    AuthorizationException,
                    ExitTimeoutException,
                    InvalidJobException,
                    NoActiveSessionException,
                    OutOfMemoryException,
                    InvalidArgumentException,
                    InternalException);
```

Parameters

`jobList` - the list of names for the jobs to synchronize.
`timeout` - the maximum number of seconds to wait.
`dispose` - specifies how to treat reaping information.

Exceptions

- DrmCommunicationException – unable to communicate with the DRMS.
- AuthorizationException – the user does not have permission to synchronize against jobs.
- ExitTimeoutException – the call was interrupted before all given jobs finished.
- InvalidJobException – the job id does not represent a valid job.
- NoActiveSessionException – the session has not been initialized or *exit()* has already been called.
- OutOfMemoryException – the DRMAA implementation does not have enough free memory to perform the operation.
- InvalidArgumentException – an argument value is invalid.
- InternalException – an error has occurred in the DRMAA implementation.

6.3.10 wait

This method SHALL wait for a job with *jobName* to finish execution or fail. If *JOB_IDS_SESSION_ANY* is provided as the *jobName*, this method SHALL wait for any job submitted during this DRMAA session up to the moment *wait()* is called. This method is modeled on the wait3 POSIX routine. Only one invocation of the *wait()* method for a given job id MAY succeed. The others MUST throw an *InvalidJobException*.

Kommentar: What happens when I call wait(ANY) and there are no running or completed jobs? (see tracker #1400)

The *timeout* value SHALL be used to specify the desired behavior when a result is not immediately available. The constant value *TIMEOUT_WAIT_FOREVER* may be specified to wait indefinitely for a result. The constant value *TIMEOUT_NO_WAIT* may be specified to return immediately if no result is available. Alternatively, a number of seconds may be specified to indicate how long to wait for a result to become available.

If the call exits before timeout, either the job has been waited on successfully or there was an interrupt. If the invocation exits on timeout, an *ExitTimeoutException* SHALL be thrown or a corresponding error code returned if exceptions aren't supported. The caller should check system time before and after this call in order to be sure how much time has passed.

The method SHALL reap job data records on a successful call, so any subsequent calls to *wait()* SHALL fail, throwing an *InvalidJobException*, meaning that the job's data record has been already reaped. This exception is the same as if the job were unknown. (The only case where *wait()* MAY be successfully called on a single job more than once is when the previous call to *wait()* timed out before the job finished.)

When successful, the resource usage information for the job SHALL be provided as a `Dictionary` of usage parameter names and their values in the returned job info. The values contain the amount of resources consumed by the job and are implementation defined.

```
JobInfo wait(      in string jobName,
                   in long long timeout)
         raises (  DrmCommunicationException,
                   AuthorizationException,
                   NoResourceUsageException,
                   ExitTimeoutException,
                   InvalidJobException,
                   NoActiveSessionException,
                   OutOfMemoryException,
                   InvalidArgumentException,
                   InternalException);
```

Parameters

`jobName` - the id of the job for which to wait.
`timeout` - the maximum number of seconds to wait.

Returns

This method SHALL return the resource usage and status information as *JobInfo* instance.

Exceptions

- DrmCommunicationException – unable to communicate with the DRMS.
- AuthorizationException – the user does not have permission to wait for a job.
- NoResourceUsageDataException – the resource usage information for the given job is unavailable.
- ExitTimeoutException – the call was interrupted before the given job finished.
- InvalidJobException – the job id does not represent a valid job.
- NoActiveSessionException – the session has not been initialized or *exit()* has already been called.
- OutOfMemoryException – the DRMAA implementation does not have enough free memory to perform the operation.
- InvalidArgumentException – an argument value is invalid.
- InternalException – an error has occurred in the DRMAA implementation.

6.3.11 jobProgramStatus

The *jobProgramStatus()* method SHALL return the program status of the job identified by *jobName*. The possible values returned from this method are:

- JobProgramState:UNDETERMINED: process status cannot be determined,
- JobProgramState:QUEUED_ACTIVE: job is queued and active,
- JobProgramState:SYSTEM_ON_HOLD: job is queued and in system hold,
- JobProgramState:USER_ON_HOLD: job is queued and in user hold,
- JobProgramState:USER_SYSTEM_ON_HOLD: job is queued and in user and system hold,
- JobProgramState:RUNNING: job is running,
- JobProgramState:SYSTEM_SUSPENDED: job is system suspended,
- JobProgramState:USER_SUSPENDED: job is user suspended,
- JobProgramState:USER__SYSTEM_SUSPENDED: job is user and system suspended,
- JobProgramState:DONE: job finished normally, and
- JobProgramState:FAILED: job finished, but failed.

The DRMAA implementation MUST always get the status of the job from the DRM system unless the status has already been determined to be *FAILED* or *DONE* and the status has been successfully cached. Terminated jobs SHALL return a *FAILED* status.

```
JobProgramState jobProgramStatus(in string jobName)
          raises (  DrmCommunicationException,
                    AuthorizationException,
                    InvalidJobException,
                    NoActiveSessionException,
                    OutOfMemoryException,
                    InvalidArgumentException,
                    InternalException);
```

Parameters

`jobName` - the id of the job whose status is to be retrieved.

Returns

The *jobProgramStatus()* method SHALL return the program status.

Exceptions

- DrmCommunicationException – unable to communicate with the DRMS.
- AuthorizationException – the user does not have permission to query for a job's status.
- InvalidJobException – the job id does not represent a valid job.
- NoActiveSessionException – the session has not been initialized or *exit()* has already been called.
- OutOfMemoryException – the DRMAA implementation does not have enough free memory to perform the operation.
- InvalidArgumentException – an argument value is invalid.
- InternalException – an error has occurred in the DRMAA implementation.

6.3.12 contact

If this attribute is read before the first call to the *init()* method, then it SHALL return a string containing a comma-delimited list of default DRMAA implementation contacts strings. A contact string represents a specific installation of a specific DRM system, e.g. a Condor central manager machine at a given IP address.
If the value of the attribute is queried after a successful call to *init()*, this attribute SHALL contain the contact String for the DRM system to which the session is attached.
The returned Strings are always implementation dependent and SHOULD NOT be interpreted by the application.

```
readonly attribute string contact;
```

6.3.13 version

This attribute SHALL contain a *Version* instance containing the major and minor version numbers of the DRMAA library. This attribute may not be read before *init()* has been called.

```
readonly attribute Version version;
```

6.3.14 drmsInfo

If the value of this attribute is read before the first successful call to the *init()* method, this attribute SHALL return a string containing a comma-delimited list of DRM system identifiers. A DRM system identifier denotes a specific type of DRM system, e.g. Sun Grid Engine.
If the value is read after *init()*, this attribute SHALL contain the selected DRM system identifier.
The returned Strings are implementation dependent and SHOULD NOT be interpreted by the application.

Kommentar: In OO world this should be a StringList type instead. Same for the other attributes that return comma-separated strings. We could also split up the functionality before and after init() in two attributes.

```
readonly attribute string drmsInfo;
```

6.3.15 drmaaImplementation

If the value of this attribute is read before the first successful call to *init()*, this attribute SHALL return a string containing a comma-delimited list of DRMAA implementations. A DRMAA implementation string denotes a specific version of a DRM system, e.g. Condor v6.6.
If read after *init()*, this attribute SHALL contain the selected DRMAA implementation.
The returned Strings are implementation dependent and SHOULD NOT be interpreted by the application.

```
readonly attribute string drmaaImplementation;
```

Kommentar: With DRMAA 1.0 drmaa_wait() / drmaa_synchronize() can be used to wait for session jobs to finish.We might want to have a corresponding means that would allow to synchronize with jobs' start. (see tracker #890)

Kommentar: We need to discuss about the possibility for multiple concurrent DRMAA sessions (see tracker #1396).

7 Annex

7.1 Correlation of DRMAA error codes and exceptions

The following table shows how the error codes defined in the Distributed Resource Management Application API Specification 1.0, correlated to the exceptions in this specification.

Error Code Name (DRMAA_ERRNO_...)	*Exception Name*
SUCCESS	none
INTERNAL_ERROR	InternalException
DRM_COMMUNICATION_FAILURE	DrmCommunicationException
AUTH_FAILURE	AuthorizationException
INVALID_ARGUMENT	InvalidArgumentException
NO_ACTIVE_SESSION	NoActiveSessionException
NO_MEMORY	OutOfMemoryException
INVALID_CONTACT_STRING	InvalidContactStringException
DEFAULT_CONTACT_STRING_ERROR	DefaultContactStringException
DRMS_INIT_FAILED	DrmsInitException
ALREADY_ACTIVE_SESSION	AlreadyActiveSessionException
DRMS_EXIT_ERROR	DrmsExitException
INVALID_ATTRIBUTE_FORMAT	InvalidAttributeFormatException
INVALID_ATTRIBUTE_VALUE	InvalidAttributeValueException
CONFLICTING_ATTRIBUTE_VALUES	ConflictingAttributeValuesException
TRY_LATER	TryLaterException
DENIED_BY_DRM	DeniedByDrmException
INVALID_JOB	InvalidJobException
RESUME_INCONSISTENT_STATE	ResumeInconsistentStateException
SUSPEND_INCONSISTENT_STATE	SuspendInconsistentStateException
HOLD_INCONSISTENT_STATE	HoldInconsistentStateException
RELEASE_INCONSISTENT_STATE	ReleaseInconsistentStateException

Error Code Name (DRMAA_ERRNO_...)	*Exception Name*
EXIT_TIMEOUT	ExitTimeoutException
NO_RUSAGE none	NoResourceUsageException InvalidJobTemplateException
none	UnsupportedAttributeException

The DRMAA_ERRNO_SUCCESS code clearly does not need to be represented as an exception. This specification introduces two new exceptions which have no error code correlatives. The *InvalidJobTemplateException* is used to indicate that the job template instance currently being used is not valid. This may be, for example, because it has already been deleted via *Session::deleteJobTemplate()*. The *UnsupportedAttributeException* is used to indicate that for the current DRMAA implementation the accessed attribute of a job template is unsupported.

7.2 Correlation of DRMAA and OO job template attributes

The following table shows the relation between DRMAA attribute names and the attribute names used in this document in a job template.

DRMAA Attribute	*OO Attribute*
drmaa_remote_command	remoteCommand
drmaa_v_argv	args
drmaa_js_state	jobSubmissionState
drmaa_v_env	jobEnvironment
drmaa_wd	workingDirectory
drmaa_job_category	jobCategory
drmaa_native_specification	nativeSpecification
drmaa_v_email	email
drmaa_block_email	blockEmail
drmaa_start_time	startTime
drmaa_job_name	jobName
drmaa_input_path	inputPath
drmaa_output_path	outputPath

DRMAA Attribute	*OO Attribute*
drmaa_error_path	errorPath
drmaa_join_files drmaa_transfer_files	joinFiles transferFiles
drmaa_deadline_time	deadlineTime
drmaa_wct_hlimit	hardWallclockTimeLimit
drmaa_wct_slimit	softWallclockTimeLimit
drmaa_run_duration_hlimit drmaa_run_duration_slimit	hardRunDurationLimit softRunDurationLimit

8 Security Considerations

Security issues are not discussed in this document. The scheduling scenario described here assumes that security is handled at the point of job authorization/execution on a particular resource.

9 References

[OMG IDL] Object Management Group. Common Object Request Broker Architecture: Core Specification, Chapter 3, March 2004

[RFC 2119] S. Bradner. RFC 2119 – Key words for use in RFCs to Indicate Requirement Levels, March 1997

10 Author Information

Roger Brobst
rbrobst@cadence.com
Cadence Design Systems, Inc
555 River Oaks Parkway
San Jose, CA 95134

Andreas Haas
andreas.haas@sun.com
Sun Microsystems GmbH
Dr.-Leo-Ritter-Str. 7
D-93049 Regensburg
Germany

Hrabri L. Rajic
hrabri.rajic@intel.com
Intel Americas Inc.
1906 Fox Drive
Champaign, IL 61820

Daniel Templeton
dan.templeton@sun.com
Sun Microsystems GmbH
Dr.-Leo-Ritter-Str. 7
D-93049 Regensburg
Germany

Peter Tröger
peter.troeger@hpi.uni-potsdam.de
Hasso-Plattner-Institute
University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3
D-14482 Potsdam
Germany

11 Intellectual Property Statement

12 Full Copyright Notice