Configuration Description, Deployment, and Lifecycle Management

Component Model Specification
Draft 2005-04-18

Status of this Memo

This document provides information to the community regarding the specification of the Configuration Description, Deployment, and Lifecycle Management (CDDLM) Language. Distribution of this document is unlimited. This is a DRAFT document and continues to be revised.

Abstract

Successful realization of the Grid vision of a broadly applicable and adopted framework for distributed system integration, virtualization, and management requires the support for configuring Grid services, their deployment, and managing their lifecycle. A major part of this framework is a language in which to describe the components and systems that are required. This document, produced by the CDDLM working group within the Global Grid Forum (GGF), provides a definition of the CDDLM component model and the process whereby a Grid Resource is configured, instantiated, and destroyed.

GLOBAL GRID FORUM
office@ggf.org
www.ggf.org

Full Copyright Notice

Copyright © Global Grid Forum (2004). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the GGF or other organizations, except as needed for the purpose of developing Grid Recommendations in which case the procedures for copyrights defined in the GGF Document process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the GGF or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and THE GLOBAL GRID FORUM DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Intellectual Property Statement

The GGF takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the GGF Secretariat.

The GGF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this recommendation. Please address the information to the GGF Executive Director (see contact information at GGF website).

Table of Contents

Table of Contents 3
1 Introduction 5
1.1 Notational Conventions 5
2 CDDLM-WG and the Purpose of this Document 6
2.1.1 Configuration Description Language 6
2.1.2 Component Model 7
2.1.3 Deployment API 7
3 Component Model Architecture .... **Error! Bookmark not defined.**
3.1 Introduction 7
3.2 Core Concepts and Terminology 8
3.2.1 Deployable Systems 9
3.3 Deploying Components 10
3.4 CDDLM State Transition Model 11
3.4.1 State Transition Events .... **Error! Bookmark not defined.**
3.4.2 State and State Extension Representation 13
4 Component Model 14
4.1 Deployment Components 14
4.1.1 ComponentReference 15
4.1.2 CodeBase 15
4.1.3 CommandPath 16
4.2 Component Delegates .... **Error! Bookmark not defined.**
4.2.1 Delegate 16
4.3 Deployment systems 17
4.4 Deployment Events .... **Error! Bookmark not defined.**
4.4.1 Event Format 26
4.4.2 Event Registration 26
Within the CDL document declaring components, event notifications will also be declared. All event declarations will take the same form. A deployment component will contain a CDL property that declares which event to register a notification for and the target of the notification. The component model requires that a deployment component MUST support the automatic registration of these event handlers at the time of component instantiation 26
4.4.3 Property Change Notifications 28
4.5 Deployment Control Flow .... **Error! Bookmark not defined.**
4.5.1 Sequence .... **Error! Bookmark not defined.**
4.5.2 Flow .... **Error! Bookmark not defined.**
4.5.3 Wait .... **Error! Bookmark not defined.**
4.5.4 Switch .... **Error! Bookmark not defined.**
5 Component Properties and Operations 18
5.1.1 Component Properties 19
5.2 Basic Component Operations 21
5.2.1 WS-Resource Properties 21
5.2.2 State transition operations 22
5.2.3 Maintenance operations 22

All tasks will execute synchronously and in response to a task command, a response MUST be generated. A fault is NOT expected to be transmitted from this request, unless it is a fault of the communications or security infrastructure. The result of the task follows that of the input with the addition of an unrequired integer status code. . 23
5.3 Component-specific message sets and messages 31
6 Reference Resolution 37
7 Faults .... **Error! Bookmark not defined.**
7.1 StateActionFault 23
7.2 OnFault 28
8 Security Considerations 37
9 Implementation Notes 38
9.1.1 Loose Coupling 38
9.1.2 Declarative Configuration rather than Direct Invocation 38
9.1.3 Minimal use of Distributed Object concepts 39
9.1.4 Do not rely on schema validation 39
9.1.5 Pass data in messages, not as references to resources 39
9.1.6 Make effective use of the proposed `ImplementsMessageSet` message 40
10 Editor Information 40
11 Acknowledgements 41
References 42
Appendix A – Component Object Definitions 43
A.1 XML Schema 43
A.2 WSDL 1.1 47
A.4 Topic Space 49

1 Introduction

Deploying a complex, distributed service presents many challenges related to service configuration and management. These range from how to describe the precise, desired configuration of the service, to how we automatically and repeatably deploy, manage and then remove the service. This document addresses the description challenges, while other challenges are addressed by the follow-up documents . Description challenges include how to represent the full range of service and resource elements, how to support service "templates", service composition, correctness checking, and so on. Addressing these challenges is highly relevant to Grid computing at a number of levels, including configuring and deploying individual Grid Services, as well as composite systems made up of many co-operating Grid Services.

1.1 Notational Conventions

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

The following namespaces are used in this document:

xs	http://www.w3.org/2000/10/XMLSchema
wsa	http://schemas.xmlsoap.org/ws/2004/08/addressing
wsrf-bf	http://www.ibm.com/xmlns/stdwip/web-services/WS-BaseFaults
wsrf-rp	http://www.ibm.com/xmlns/stdwip/web-services/WSResourceProperties
wsrf-rl	http://www.ibm.com/xmlns/stdwip/web-services/WSResourceLifetime
wsrf-nt	http://www.ibm.com/xmlns/stdwip/web-services/WSBaseNotification
wstop	http://www.ibm.com/xmlns/stdwip/web-services/WSTopics
muws-p1-xs	http://docs.oasis-open.org/wsdm/2004/12/muws/wsdm-muws-part1.xsd
muws-p2-xs	http://docs.oasis-open.org/wsdm/2004/12/muws/wsdm-muws-part2.xsd
mows-xs	http://docs.oasis-open.org/wsdm/2004/12/mows/wsdm-mows.xsd
cdl	http://www.gridforum.org/2004/12/CDDLM/XML-CDL/1.0
cmp	http://www.gridforum.org/cddlm/components/2005/02
api	http://www.gridforum.org/cddlm/serviceAPI/2004/10/11

[TODO: Define the grammar used in more detail]
When defining operations, this specification uses pseudo-schema to describe the input and,if appropriate, output messages. A full XML Schema and WSDL description of all elements and operations are available in Appendix A of this specification.

Sections 4 and 5 and appendices A.1 through A.4 containing the XML Schema (XS) of the component model and the WSDL component interface are *normative* specifications. All UML diagrams found are non-normative, and meant to serve only as illustrations of the CDDLM concepts.

[TODO: Copy all italicized items into here]

Further, the following terms are defined and used throughout this document:
Resource –
Component
Language Component
Deployment Component -
Component Delegate -
Service -
Service Endpoint -
System -
Deployment Graph –
Portal EPR -
System EPR -
Component EPR -

2 CDDLM-WG and the Purpose of this Document

The CDDLM WG addresses how to: describe configuration of services; deploy them on the Grid; and manage their deployment lifecycle (instantiate, initiate, start, stop, restart, etc.). The intent of the WG is to gather researchers, developers, practitioners, and theoreticians in the areas of services and application configuration, deployment, and deployment life-cycle management and to explore the community need for a broader effort in this area. The target of the CDDLM WG is to come up with the specifications for CDDML a) language, b) component model, and c) deployment API. This document represents the component model specification. This specification is derived from the SmartFrog framework developed at HP Labs, and other industry specifications.

This document describes the component model of the framework. It does not describe any required component libraries. The design of the system is loose to allow the binding of attributes to deployment objects to be handled by an individual implementation of the system. This document serves to outline the requirements for a software object to be deployable by the CDDLM framework.

This specification is closely related to a set of three specifications that together comprise service configuration description, deployment, and lifecycle management. All three specifications are briefly described in the following three paragraphs. For more details please refer to other two specifications.

2.1.1 Configuration Description Language

The CDDLM Configuration Description Language (CDL) is an XML-based language for declarative description of system configuration that consists of components (deployment objects) defined in the CDDLM Component Model. The Deployment API uses a deployment descriptor in CDL in order to manage deployment lifecycle of systems. The language provides ways to describe properties (names, values, and types) of components including value references so that data can be assigned dynamically with preserving specified data dependencies. A system is described as a hierarchical structure of components. The language also provides prototype-based template functionality (i.e.,

prototype references) so that the user can describe a system by referring to component descriptions given by component providers.

2.1.2 Component Model

The CDDLM Component Model outlines the requirements for creating a deployment object responsible for the lifecycle of a deployed resource. Each deployment object is defined using the CDL language and mapped to its implementation The deployment object provides a WS-ResourceFramework (WSRF) compliant "Component Endpoint" for lifecycle operations on the managed resource. The model also defines the rules for managing the interaction of objects with the CDDLM Deployment API in order to provide an aggregate, controllable lifecycle and the operations which enable this process.

2.1.3 Deployment API

The deployment API is the WSRF-based SOAP API for deploying applications to one or more target computers. Every set of computers to which systems can be deployed hosts one or more "Portal Endpoints", WSRF resources which provide a means to create new "System Endpoints". A System Endpoint represents a deployed system. The caller can upload files to it, then submit a deployment descriptor for deployment. A System Endpoint is effectively a component in terms of the Component Model specification -it implements the properties and operations defined in that document. It also adds the ability to resolve references within the deployed system, enabling remote callers to examine the state of components with it.

3 Components in the CDDLM Architecture

3.1 Introduction

A prerequisite for understanding this document is to have read the CDDLM language document(s), which introduce a tree-like representation of configurable components as a way of describing the pieces of a distributed application. Using the description of the application, a system definition can be provided to an instance of the CDDLM Deployment API . It is the responsibility of this instance to realize the deployment through instantiation and control of CDDLM compliant deployment components. This basic relationship can be understood from this simple figure:

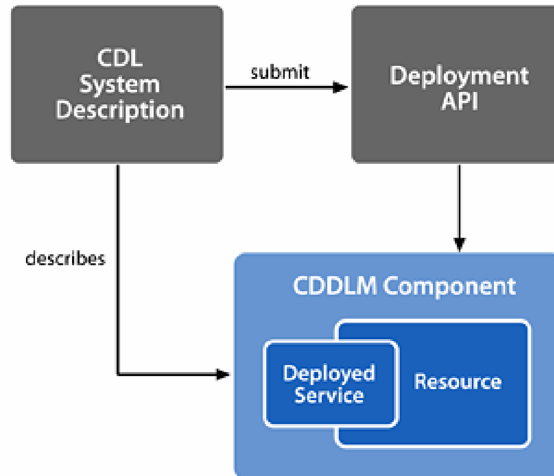

Figure 1. The CDDLM Deployment Framework.

The CDDLM component model is not intended to be a general purpose model for systems or application management. It is limited in scope to address the problems of deploying and managing the deployed state of *resources*. In the context of CDDLM, a resource is considered to be the hardware, software and other configurable items required to execute the distributed application being managed.

The basic principles of the CDDLM framework are described in the CDDLM Foundation Document. Additionally, the reader should consult the CDDLM Deployment API document in order to understand the basic deployment capabilities of the CDDLM framework.

3.2 Core Concepts and Terminology

The CDDLM component model represents a means to group a hierarchy of services within a deployment container. This container provides a wrapper for the services to aid in their deployment and management. In this containment hierarchy there may be a many-to-many relationship among services and actual instantiated resources. This hierarchy is not intended to reflect the organization of those managed resources. Its purpose is to capture the runtime deployment dependencies, operational flow and state of those resources.

At deployment time, a service will be resolved by the CDDLM implementation to determine its composition and membership within the active Grid Fabric. The basic unit of deployment is a *component*. The service to be deployed will be a part of this component in some way. There is no restriction to what a component may contain. It is simply a unit of deployment, and will imply that there is a CDDLM object which bounds its contents.

A component has two analogues. A *language component* is a configurable component along with its property list as defined in the CDL language document; it is a declaration of what the user wishes deployed, and how they want it configured. A *deployment component* is a code library that provides the implementation of language components. When the CDDLM infrastructure needs to deploy an application, it must locate and instantiate deployment components for every language component that it wishes to instantiate. These deployment components will be moved through their lifecycle, and so bring up the system.

A CDDLM *component object* is a specific instance of a deployment component. A deployment component can be instantiated multiple times in the same document or system, as well as in multiple, separate systems. The component object will correlate to a component within the system tied to one or more allocated resources. A deployment component will have one component object for each resource or group of resources that it is declared to manage.

3.2.1 Deployable Systems

Groups of components can be organized to form a deployable *system*. A system will be comprised of one or more components each implementing some portion of a complete Grid Service or prerequisite, or individually implementing one or more *services*. Each service will provide access to its implementation through one or more *endpoints*, either WSRF endpoints or standard WS-I endpoints, as defined in the services own WSDL. All such endpoints will be presumed to be valid Endpoint References (EPRs) as defined by the WS-Addressing specification. The relationships between services and components are modeled in the language and exposed in order to facilitate these compositions.

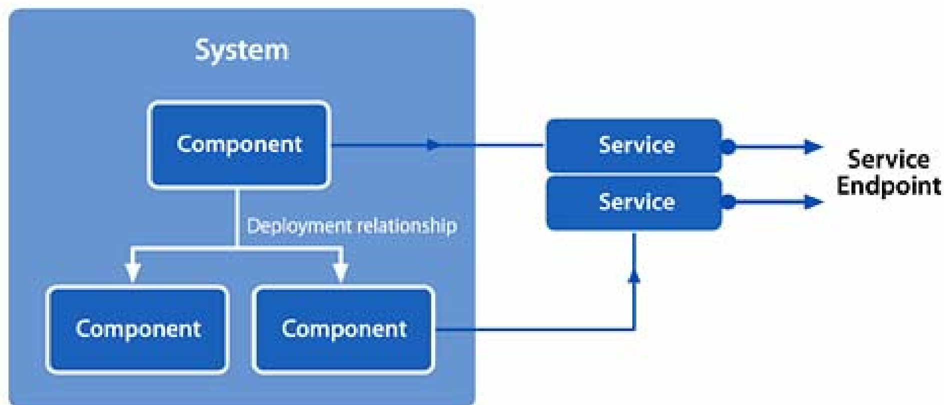

Figure 2. A Deployable System.

Not all services will be complete software applications. A service endpoint may exist within a larger set of an application server, for example. A service may also be dependent upon the configuration of a more fundamental portion of software such as an operating system or patch. Or there may be a pre-existing service that does not implement the CDDLM interfaces. In this manner, the CDDLM component model

enables services to be "wrapped" within a component container that is responsible for implementation of the defined component interfaces.

In many cases, the services that are deployed do not meet the manageability requirements of its environment. The CDDLM component model allows a component to wrap an implemented service and provide a level of manageability that otherwise would not exist. These manageability functions implement CDDLM specific interfaces as well as those required by the WSDM specifications. Other interfaces could be provided as the implementation dictates.

Though the conceptual diagram above depicts a system as shown hierarchically, there is no requirement by CDDLM for components to be explicitly organized in this way. A system is just a loose confederation of components needed to deploy some set of endpoints. The following UML diagrams the conceptual relationship of these entities.

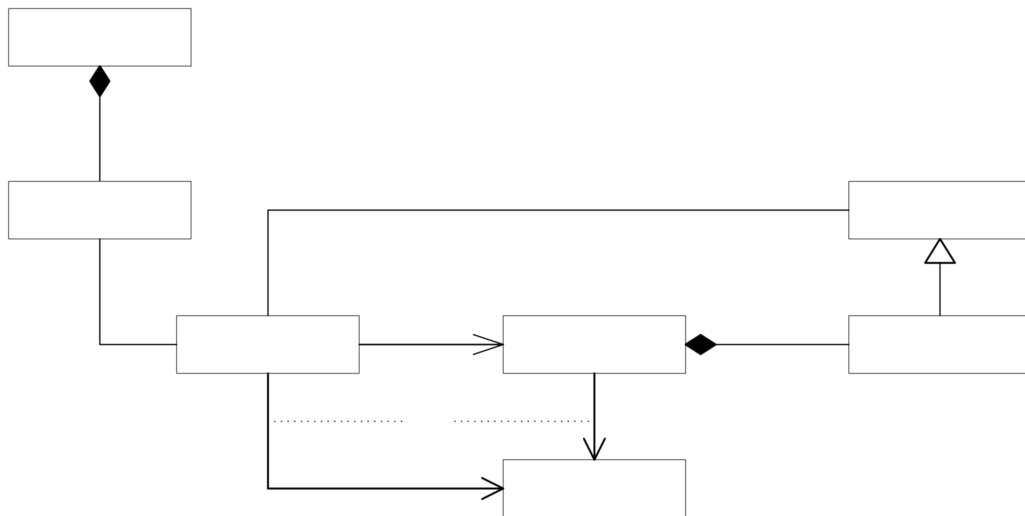

Figure 3. Component Model Conceptual Diagram.

3.3 Deploying Components

In order to deploy, the CDDLM framework is sent a description of the system in a supported language through a node implementing the Deployment API. This node is addressed via a well known WS-Addressing Endpoint Reference (EPR) [WS-A], a *portal EPR*. The run time of this node parses the description to produce an internal representation of the system. This will be a (annotated) tree of language components -a *deployment graph*. This graph describes what must be deployed, and what the interdependencies are.

Each system to be deployed is instantiated through the creation of a WS-Addressing Endpoint Reference to a resource responsible for its operation. This *system EPR* represents the aggregate state of the collection of deployed components for a particular system instance. This EPR provides a WS-ResourceFramework resource with all of its

methods and properties exposed using the WSRF embodiment of the service endpoint. The complete description of using the *system EPR* is described in the CDDLM Deployment API.

The resource responsible for the *system EPR* or *portal EPR* will parse the CDL definition to map language components to deployment components. The parsing phase produces a complete XML CDL representation of the deployment graph. All internal use of documents and languages by the component model are based on the XML CDL specification, no other languages are supported. The deployment graph will be segmented and for each deployment component identified a WS-Addressing endpoint reference, a *component EPR,* is assigned to each deployment component.

A *component EPR* is the primary interface to manage a deployment component. Component EPRs are references to the resource which is an instance of the corresponding deployment component. This EPR will also rely on the mechanisms defined by the WS-ResourceFramework. There is no requirement of this component model or by CDDLM that a component map to only a single resource, or that the address of the component endpoint correlate in any way to the deployment location of the resource. Thus, the embodiment of "WS-Addressing Using Reference Properties" MUST be followed. All message exchanges with a component will include the ReferenceProperties of the WS-Resource qualified component EPR.

Once deployed, the component EPRs are used internally for components to communicate with the system EPR and with each other. The node responsible for the system EPR is able to send lifecycle commands to component EPRs and receive lifecycle events. The system EPR provides endpoint resolution services to components in order to enable this intercommunication. This resolution service allows components to translate CDL language references to another component into the EPR of that component. This resolution interface is described in detail in the Deployment API specification.

3.4 Component Lifecycle

The basic lifecycle of components, and thus the resources being managed, is defined by a state machine that is implemented by all component objects within CDDLM. The CDDLM lifecycle is restricted, in that it supports the deployment processes used by the framework and models only the deployment state of the system, not its operational characteristics. For example, once a CDDLM component has entered the running state, CDDLM considers it to be available. In reality, the service that is being deployed can be operative but have suffered some set of internal failures.

Each deployment component independently represents the state of the deployed resource which it is managing. The system as a whole must also represent a reasonable depiction of the overall state of many components. In this state machine, there is no representation of transitional state, only of arrival at new states. The core lifecycle is defined by the states, allowed transitions and operations shown in the diagram in below.

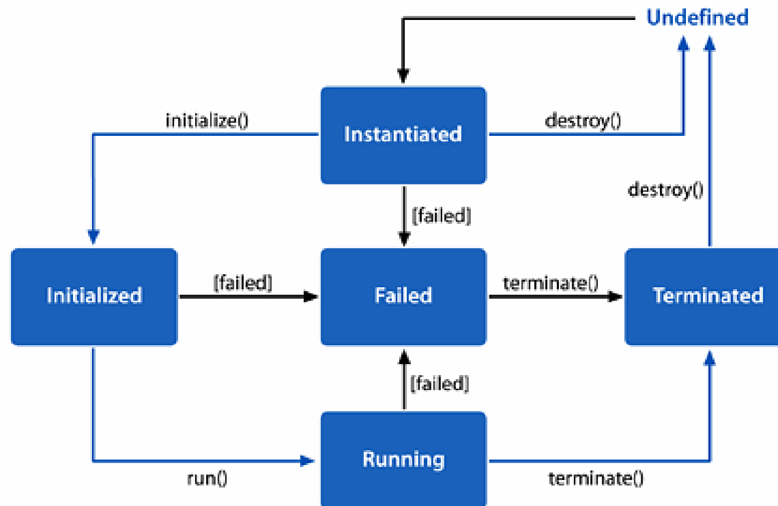

Figure 4. CDDLM Component Lifecycle Model.

The *undefined* state represents any state in which the component is not known to have an instance running or instantiated in the system. In the undefined state, the assets of the component may or may not have been deployed from the rest of the framework. There is no requirement by the component model that the assets of the system be stored or removed during any lifecycle operation.

When a component is created, it will make the transition from undefined to *instantiated*. During this time, the component and its deployment package will be deployed to the appropriate system, the component instantiated and any operations will be performed that are part of the component's instantiation process. This state also presumes that whatever activation is required in order for the WSRF resource identifier of the component to be valid has been performed. Operations on the endpoint reference can now be performed. As shown in the diagram above, the `initialize()` and `destroy()` state change commands are supported in this state.

The next state of the system is the *running* state. This state indicates the services provided by the resources that are being deployed are available for use. As mentioned earlier, this state does not indicate any information regarding the operational capabilities of the service, only that it has completed initialization and not failed.

At any time, state actions may not complete correctly or the service itself may fail. In response to these failures, the component will transition to the *failed* state. The component may remain active in the system, but its managed resource is presumed to no longer be operational.

Once a component is running or has failed, it must be terminated to stop its services. The *terminated* state represents a state where a component is no longer running and cannot be returned to the running state without redeployment. This state, however, does not eliminate the resource from the system. Upon invocation of the `destroy()` command, the

component will return to the undefined state and the corresponding resource becomes invalid.

In a system with multiple components, the lifecycle of the whole system is defined by the relationships between the individual component lifecycles. The hierarchy of the system defines relationships where related components' lifecycles are linked. The component model and the CDL language help define explicit semantics for guiding lifecycle transitions.

3.4.1 State and State Extension Representation

It is also desired for the lifecycle model to be extensible at the component level. The core lifecycle described above is required for implementation of CDDLM. However, CDDLM will enable services to declare service specific lifecycle extensions that are declared for a particular component and operate within the basic deployment lifecycle model defined herein. In this way, a service can be deployed and managed simply, but also define internal states relevant to other services capable of consuming its extended information.

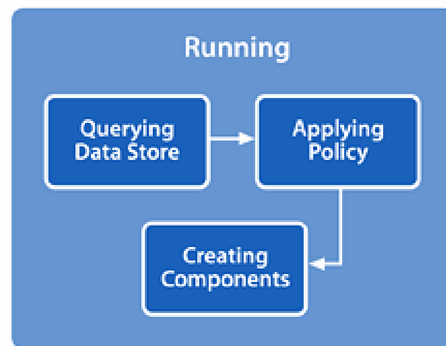

Figure 5. Extended States.

In the example above, a service is currently known to CDDLM and has been created and is running. If it is sent a command such as "Refresh Data", it may externally remain in the running state. However, internally it may go through several state transitions in order to execute the command. By declaring these states in a resource property document, its states can be tracked and understood through the basic component model interfaces.

This extension mechanism does not require the definition of the state machine or state transitions to the CDDLM framework. This would require implementation of a formal calculus within the CDL language, and a secondary state machine within the framework. Instead, the lifecycle model is implemented in the component model using the WSDM MUWS State capability. This allows the state of the system to be exposed as a WSRF resource property with a simple and flexible model for extension.

4 Components

This section describes the model for deployment components within the CDDLM framework: what they are, how they work, and how they integrate with the framework. The CDDLM framework requires a concrete object that can be instantiated in order to perform a deployment action. This object's purpose is to abstract the basic framework from the details of the composition of what is to be deployed and its specific, required deployment steps.

Components are represented in the CDL language using an abstract object model that most closely resembles languages such as Scheme: any component can have new attributes added to it dynamically, while the type system is very loose. Furthermore, there is no split between classes and instances; everything is an instance that can be extended dynamically.

There also does not exist a mapping between every component instance in the language-level model and implementation model. That would be neither sensible nor possible. Instead, any component in the language can be entirely abstract. This has benefit for providing templates and parameterization. Language components can also be used to provide higher level behaviors to coordinate deployments. For a list of supported language component behaviors, consult the CDL Language Specification Document.

4.1 Deployment Components

In the CDDLM system, a CDL document will contain one or more language components. In order to turn this into a deployment, the framework will need to map these language components into deployment components. When a `create()` command is sent to the Deployment API, the system will parse the CDL document and create a mapping.

At least one of the language components declared in the CDL document MUST map to a deployment component. Otherwise, no action can be taken during the deployment phases, as all concrete actions of the framework are performed against the deployment components of a system. Any component object MAY simply exist to assist with the deployment and have no actual internal objects to deploy or resources to manage. Each component object, though, MUST implement the basic interfaces defined in this specification document.

The parsed CDL document can be walked to identify deployment components and assemble a deployment graph and actions to perform in order to instantiate components. For each element that declares an asset location and means to activate the assets, an object will be created. The runtime will deploy the code contained at the source location, and attempt to instantiate the object. An example of how this is done with CDL is shown below.

```
<cdl:cdl targetNamespace=”http://example.org/webapp-template”>
<cdl:system>
<WebApplication>
  <cmp:CodeBase>http://server/file.jar</cmp:CodeBase>
```

```
  <cmp:CommandPath>com.exns.ApacheDeployer</cmp:CommandPath>
  <port>80</port>
  <hostname>www.example.org</hostname>
</WebApplication>
</cdl:system>
</cdl:cdl>
```

In this case, the `WebApplication` corresponds to an `ApacheDeployer` deployment component located within the deployment container `file.jar` at the remote web server location.

Once deployment components have been identified, the processing is very platform specific. A Java deployment component will be activated within some JVM to become a component object. An executable will be run on its host. A COM or other object will be created within some host specific to the platform. It is important that during the resolution and instantiation process that the CDDLM framework is able to bind the component to the correct runtime or host. Further parameterization of the host requirements should be given in the CDL for the component.

There are a base set of component properties that define a deployment component within a CDL document. Each one of these properties, if declared within the CDL document, MUST be exposed as resource properties of the component as defined by the WS-ResourceProperties specification
.

4.1.1 ComponentReference

The component endpoint reference (EPR) SHOULD be specified within the CDL document. This element definition enables the CDDLM framework to identify a target resource's endpoint. During the resolution phase of parsing the CDL document, as described in the CDL language document, CDDLM MUST identify the target resources for the deployment and assign them unique endpoint references.

The `ComponentReference` CDL property is defined by the following XML Schema fragment. It MUST be a direct child of the language component it modifies. It cannot be multiply defined for a single component.

```
<xsd:element name="ComponentReference" type="wsa:EndpointReferenceType"
maxOccurs="1"/>
```

As the component object is responsible for one or more managed resources, its endpoint reference MUST include the ReferenceProperties child element with a stateful resource identifier. This will ensure that the component object resource is clearly identified separately from the resource it is managing.

4.1.2 CodeBase

The component code base is defined as any location of file assets at some URI. It is up to the Deployment API how to obtain and upload the file(s). Any CDL element which is declared as part of the system description, a child of the `<cdl:system>` element, is potentially a deployment component. If such an element contains the `CodeBase` property

as a direct child of the element, that element MUST be considered a deployment component. If the same element does not contain a `CommandPath` property to indicate how to activate the component, an error MAY result.

The `CodeBase` CDL property is defined by the following XML Schema fragment. This property can be defined as many times as needed to indicate all of the possible sources of deployment assets.

```
<xsd:element name="CodeBase" type="xsd:anyURI" minOccurs="1"
maxOccurs="unbounded"/>
```

4.1.3 CommandPath

The command path is some referenceable path indicating to the Deployment API how and/or what to instantiate in order to realize this deployment component. This path is defined as an opaque string as it may be any one of a number of things such as a Java class path, an executable name, a COM object name or GUID, or any other platform specific instantiation name. The `CommandPath` property can also include sub-elements that can be used as arguments or other augmentation of the base path.

The `CommandPath` CDL property is defined as a `cmp:commandPathType` element which is defined by the following XML Schema fragment.

```
<xsd:element name="CommandPath" type="cmp:commandPathType"/>
<xsd:complexType name="commandPathType">
  <xsd:sequence>
    <xsd:element name="path" type="xsd:anyURI"/>
    <xsd:element name="args" type="xsd:anyType" minOccurs="0"
maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:simpleType>
```

When a component is instantiated, the CDL document may define multiple properties in its property list. An implementer of this component model MAY choose to pass these arguments through the argument parameters, or pass the entire CDL document or relevant fragment. It is RECOMMENDED that these properties are set by the Deployment API using the WSRF SetResourceProperties operation defined by the component model.

4.2 Component Delegates

A derived type of component is a *component delegate*, a deployment component that manages a set of child components on behalf of the system. Component delegates extend the basic functionality of a component by aggregating the lifecycle and operations of components that are declared to be their children. Within the deployment, a component delegate presents a single endpoint and resource for management. However, internally all component children MUST still present a full component endpoint for management by the delegate.

In order to create a component delegate, you will simply declare that the component is a delegate. There are no attributes or children of this declaration. However, within the CDL document, a component delegate will need its children defined. Any CDL element which is identified as a deployment component will have a property list defining the

parameters of the component. The element may also have child elements as part of the property lists. Within this hierarchy, if the element is declared as delegate, any child element which is also a deployment component will be a child of the component delegate. The following XML Schema fragment shows a simple declaration of the Delegate CDL property.

```
<xsd:element name="Delegate" type="xsd:string" nillable="true">
```

Within a CDL document, a declaration may look as follows, with a database server declared as a child of the web server component delegate:

```
<WebApplication>
  <cmp:CodeBase>http://server/file.jar</cmp:CodeBase>
  <cmp:CommandPath>com.exns.ApacheDeployer</cmp:CommandPath>
  <cmp:Delegate/>
  <port>80</port>
  <hostname>www.example.org</hostname>
  <OracleServer cdl:extends="DBServer">
    <cmp:CodeBase>http://server/oracle.zip</cmp:CodeBase>
    <cmp:CommandPath>./orainst/install.sh</cmp:CommandPath>
    <username>oracle</username>
    <password>sa</password>
    <tnsname>server.exns.com</tnsname>
  </OracleServer>
</WebApplication>
```

If a component is declared to be a component delegate, but has no children, it MUST operate as a standard component. If a component has children, but it itself is not a delegate, there is no parent-child or other relationship and the component MUST NOT enforce any such dependency.

4.3 Systems

Within a CDL document, all of the elements defined by this specification will be applied as part of normal property lists, as children of CDL configuration or system elements. This is done to keep the deployment process description within the declaration of the deployment components themselves. This will allow us to use the standard parsing features of the CDL language to our benefit. The basic component model as defined thus far and relationships are characterized in the following diagram.

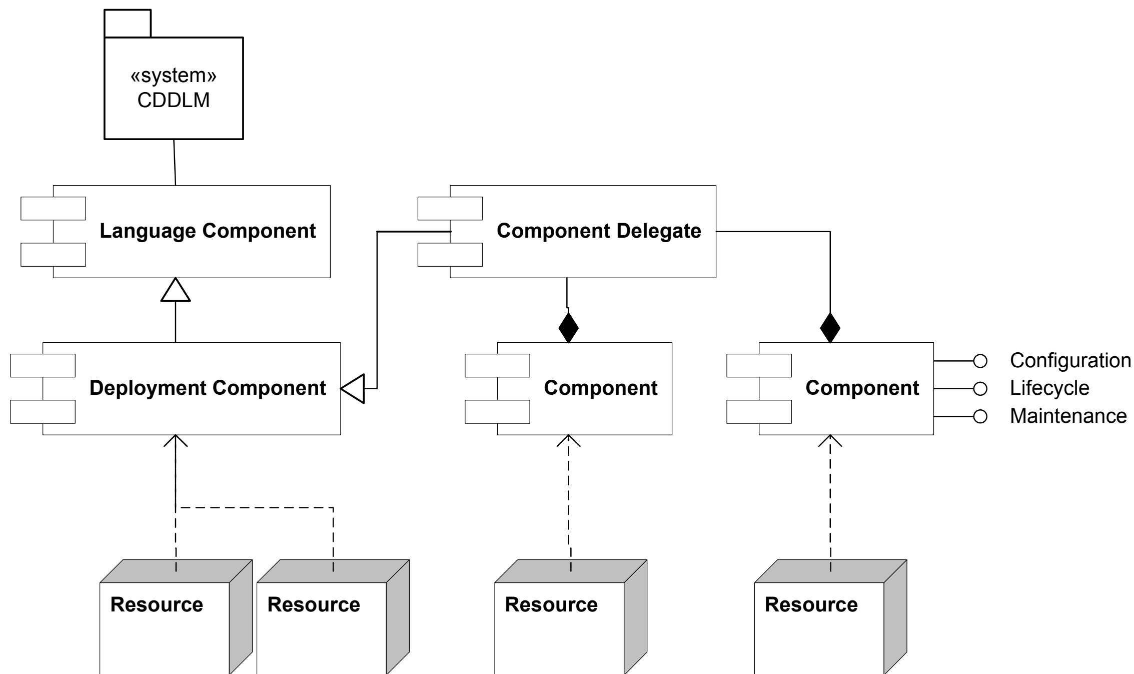

Figure 6. Basic Component Relationships.

The attribute inheritance model of CDL implies that components which derive their configurations from others must be able to override those attributes. Though this implies some derivative behavior in the language, the component model does not require the objects to be in a parent-child relationship or have some inherited behavior in its implementation. Thus there is no requirement for any form of object inheritance or other object oriented implementation.

A system is simply a bag of elements, component objects. There may be a hierarchical, peer or other relationship between objects, if deployment dependencies exist. Without dependencies the CDDLM runtime is free to deploy in any means it chooses. Any component object not related to another is a peer of all other top level components. Other than the dependent relationship, there is no other implied constraint between component objects.

4.4 Component Properties and Operations

Every deployment component managed by the CDDLM framework will support a common set of properties and operations. There may also be optional sets of messages, for use between components that have explicit knowledge of each other's message support. Callers SHOULD be able to poll endpoints to verify that a set of messages is supported before sending one or more messages from that set.

In addition to the core lifecycle functions, the components will be able to use the CDDLM framework to access their configuration information, locate other components

defined in the configuration, and potentially alter the running configuration. These interfaces will be implemented using the WSRF and WSDM specifications where appropriate, as well as companion specifications as indicated.

4.4.1 Component Properties

When a component is defined, its CDL property list defines several attributes related to its configuration. The component MUST expose each of these attributes as a resource property of the component WS-Resource. In addition to the properties associated with those defined in the CDL property list, every component has properties and actions for managing its lifecycle.

4.4.1.1 Identity

Every component must be uniquely identifiable to enable proper construction of and communication with its WSRF endpoint. The WSDM MUWS defines the Identity capability to meet this goal. A deployment component MUST implement this property and declare a unique URI as defined in the WSDM MUWS Part 1 specification. The WSDM Identity capability defines the ResourceId property for each component.

```
<xsd:element name=”muws-p1-xs:ResourceId” type=”xsd:anyURI”>
```

4.4.1.2 ComponentName

Every component will also have its own unique name which can be created from its own valid *cdl:ref* reference. This path name SHOULD be exposed to help uniquely define the component and provide confirmation of the CDDLM source in message exchanges. This attribute is defined as shown in this XML Schema fragment.

```
<xsd:element name=”ComponentName” type=”cdl:pathType”>
```

The component model does not require that this property be set in any particular manner.

4.4.1.3 ComponentStatus

Each deployment component MUST expose a state resource property which implements the MUWS State capability. To satisfy this requirement, a deployment component must contain *muws-p2-xs:State* and *muws-p2-xs:StateTransition* elements. Additionally, a deployment component may include additional information as an opaque quantity that an external consumer may be able to process. The `ComponentStatus` property will be exposed by every component object of a system.

```
<xsd:element name=”ComponentStatus” type=”cmp:componentStatusType”>
<xsd:complexType name=”componentStatusType”>
  <xsd:sequence>
    <xsd:element name="State" type="lifecycleStateType"/>
    <xsd:element ref=”LifecycleTransition”/>
    <xsd:element name="ExtendedState" type="unboundedXMLAnyNamespace"
minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:element name="ComponentStatus" type="componentStatusType"/>
```

The State element implements MUWS State by defining the lifecycle states of CDDLM as a list of *muws-p2-xs:StateType* elements. These elements are listed in the table below.

Element	Description
UndefinedState	The undefined state.
InstantiatedState	State representing the presence of a component instance.
InitializedState	State in which a component has been properly initialized.
RunningState	Operational state.
FailedState	State in which the component has failed either a lifecycle operation or its operation has failed.
TerminatedState	State in which a component instance has been terminated.

As the failed state may have been arrived at due to failures during many parts of the lifecycle, it is RECOMMENDED that the component take action to ensure the services of the resources are not available while in this state, particularly if the transition occurred from the running state.

The LifecycleTransition element is defined as a *muws-p2-xs:StateTransition* element. This state transition property SHOULD be implemented as defined by WSDM MUWS. The state TransitionIdentifier SHOULD NOT be used, but instead the PreviousState element SHOULD be used. An example XML fragment of this property could be as follows.

```
<cmp:LifecycleTransition>
  <muws-p2-xs:StateTransition Time=”2005-03-01T01:54:30Z”>
    <muws-p2-xs:EnteredState>
      <cmp:RunningState/>
    </muws-p2-xs:EnteredState>
    <muws-p2-xs:PreviousState>
      <cmp:InitializedState/>
    </muws-p2-xs:PreviousState>
  </muws-p2-xs:StateTransition>
</cmp:LifecycleTransition>
```

The final element, ExtendedState, follows the MUWS State capability which allows one to create state extensions according to the rules of MUWS Category Taxonomies. These rules MUST be followed, particularly that any extension to the state model listed in this document MUST be declared in a resource property document with a valid QName to identify the semantics of the extensions. Also, state extensions must be done by declaring a state to **contain** the states declared in this component model. A schema using this capability would define its own sub states as follows this XML Schema fragment.

```
<xsd:element name=”ApplyingPolicy”>
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:restriction base=”cmp:lifecycleStateType”>
        <xsd:sequence>
          <xsd:element ref=”cmp:RunningState”/>
        </xsd:sequence>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
```

As an example, the “Appling Policy” sub-state from the example in Section 3 may be represented by the following XML fragment.

```
<exns:ApplyingPolicy xsi:type=”cmp:lifecycleStateType>
  <cmp:RunningState/>
</exns:ApplyingPolicy>
```

4.4.1.4 Deployment properties

In order to facilitate discovery operations, a deployment component needs to provide access to its Resource Property document. The basic properties of a component are declared within the `ComponentProperties` element. A component MAY additionally support the WS-MetadataExchange specification to enable dynamic retrieval of a component’s property declarations. This element is normatively declared in the XML Schema in Appendix A.

4.4.2 Basic Component Operations

In order to access a component’s attributes and work with its management functions, a class of operations are exposed to enable these behaviors. These behaviors are expected to be used both internally by the CDDLM framework and with external systems.

4.4.2.1 WS-Resource Properties

Component properties in the component model can be directly manipulated using the WS-ResourceProperties specified mechanisms. Each component MUST implement the GetResourceProperty operation and SHOULD implement GetMultipleResourceProperties to assist in scalable access patterns.

A component or the system EPR host may choose to alter the value of a component’s property. This is most likely as the result of some event or decision step in the deployment process. For completeness, the component SHOULD support each of the Insert, Update and Delete operations of the WS-Resource.

In order to support enumeration and introspection type operations on the component’s attributes, a component MAY implement the QueryResourceProperties operation. It is RECOMMENDED that an implementation supports the XPath 1.0 query expression dialect at a minimum, additional query expression dialects MAY also be supported.

Optionally, a component EPR or the system EPR may provide a means to retrieve the component’s current WSDL which can be dynamically generated to reflect the component’s current state. That operation is not normatively defined.

Each component object MUST support lifecycle state observation. This interface will use the WSRF property query ability to retrieve the `ComponentStatus` property of the component.

4.4.2.2 State transition operations

During the life of an object, the CDDLM framework will invoke state transition actions on the object. In order to follow the WSRF recommendations, each object will provide a separate interface for each state transition request: initialize, run, terminate, and destroy. An instantiate message is unnecessary, as the `create`() operation handled by the Deployment API will have instantiated each component object.

Note that the `destroy`() operation is a WS-ResourceLifetime operation, not a CDDLM specific implementation. It is NOT RECOMMENDED to implement scheduled termination as part of the WSRF implementation for components.

Each of these messages take no parameters in their input, and return no data. If a failure occurs, a fault of type `StateActionFault` is thrown. It can also be expected that a component will transition to the failed state in this event. All of these operations MUST be performed synchronously. If CDDLM consumers wish to find out more about the state transition, it must issue a state query request.

Upon invocation of a particular state command, a component may take any appropriate action and alter its referenced state. If no action is to be taken, the component MUST at least guarantee transfer to the next state. A component SHOULD NOT report its state as the next state until the transition is complete. This conservative approach prevents reporting a successful transition followed by a failure due to problems transitioning to the requisite state.

[TODO: Add details for each operation]

4.4.2.3 Maintenance operations

It is often the case that simple maintenance activities can prolong the healthy lifespan of a long-running service. It is desired to allow CDDLM objects to provide a simple command execution infrastructure to enable periodic maintenance or cleanup of these services. This interface will presume that the object is knowledgeable in means to execute the command.

All maintenance commands are modeled as simple tasks. The CDDLM framework will make no interpretation of the commands contained in the task. Each task will be a named task for tracking by external infrastructure, as well as provide the basic content of the task. Optionally, the task can contain structured content from any derived namespace.

```
<xsd:element name="taskActionRequest">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="taskName" type="xsd:string" minOccurs="1"
maxOccurs="1"/>
      <xsd:element name="taskContents" type="unboundedXMLAnyNamespace"
minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

All tasks will execute synchronously and in response to a task command, a response MUST be generated. A fault is NOT expected to be transmitted from this request, unless it is a fault of the communications or security infrastructure. The result of the task follows that of the input with the addition of an OPTIONAL integer status code.

```
<xsd:element name="taskActionResponse">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="taskName" type="xsd:string" minOccurs="1"
maxOccurs="1"/>
      <xsd:element name="taskStatusCode" type="xsd:integer" minOccurs="0"
maxOccurs="1"/>
      <xsd:element name="taskOutput" type="unboundedXMLAnyNamespace"
minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

4.5 Faults

All of the operations defined in the component model may return faults as an indication that an error has occurred. All faults are based on the WS-BaseFaults specification. Faults are raised in response to errors at resource endpoints or in the communication infrastructure or other portions of the distributed system. Faults can be transmitted in response to operations on the component endpoint or sent as part of notification events. Service implementations may also implicitly raise SOAPFault faults, as that is inherent in most implementations.

4.5.1 Fault Categories

There is a base fault `DeploymentFault`, from which all other faults are derived. This fault class directly overrides the WS-BaseFault fault type. It adds extra information useful for debugging and managing distributed systems. This items are reflective of the lessons learned in web services development [Loughran02], namely that extra information such as hostname and process is essential for locating which process among many has failed on a clustered system.

All interfaces must declare that they raise these `DeploymentFault` instances, rather than list the specific faults. This is to provide forward extensibility. If an implementation has a fault state whose meaning matches that of the predefined fault, the predefined fault MUST be thrown. If this predefined fault has standard elements for embedded fault information, the implementation SHOULD fill them in. The implementation MAY add implementation-specific data within the `ExtraData` element of the fault, to supplement this information. This extra data MUST not add new types to the XML namespaces of this deployment data. The XML schema and semantics of this extra data should be documented.

If an existing fault type is not suitable, implementations MAY create new fault types. If an implementation creates new fault types, these MUST extend the existing fault types which operations are declared as throwing, which effectively means that they must extend `DeploymentFault`. These new faults MUST NOT change the XML schemas of

CDDLM, and they MUST be in a new namespace. The new faults and XML content SHOULD be publicly documented.

If an implementation adds new operations or properties at the existing endpoints, these new operations MAY raise whatever faults they see fit, within the constraints of the WS-BaseFault specification. Again, the implementation must not add new types to the CDDLM namespaces.

4.5.1.1 Transport faults

Transport faults will inevitably be raised as the appropriate fault for the system. For example, the Apache Axis SOAP client raises `AxisFault` faults for all SOAP events, wrapping stack trace and even HTTP Fault data within the fault as DOM elements. Microsoft .NET WSE has a similar fault class.

4.5.1.2 Relayed Faults

Relayed faults are those received by the far end and passed on. They may be WS-BaseFault Faults; HTTP error codes, SOAP faults, native language faults wrapped as SOAPFaults, or predefined deployment faults. WS-BaseFault uses fault nesting for relaying faults; however, all faults must be a derivative of WS-BaseFault. This is addressed by defining a new WS-BaseFault derivative, a `WrappedSOAPFault`. This type is actually an extension of `DeploymentFault`. This fault can nest any received SOAPFault, with an element containing the received XML data. Well-known elements in this fault data (such as the Apache Axis stack trace and HTTP fault code) should be copied into any fields in the main fault that fill the same role.

4.5.2 Fault Security

Sites offering deployment services, may, for security reasons, wish to strip out some information, such as stack trace data. Implementations should provide a means to enable such an action prior to transmitting faults to callers.

Host name and process information may be viewed as sensitive, yet again, this is exceedingly useful to operations. Implementations may provide a means to disguise this information, so that it does not describe the real hostname or process ID of a process, but instead pseudonyms that can still be used in communications with any operations team.

4.5.3 Internationalization

The WS-BaseFault specification makes no statement upon which language error descriptions are described. If an implementation can return descriptions in one language, it must use `xml:lang` attributes to indicate the language of a description. Multiple descriptions, in different languages may be included. The client application should extract the description(s) whose language is the nearest match to that of the client.

4.5.4 DeploymentFault

This type represents any fault thrown by a component or the deployment infrastructure. All endpoint operations MUST declare that they throw this fault, and must not explicitly declare any derivative faults that they may throw.

Element	Type	Description

Host	xsd:string	Hostname or pseudonym.
Process	xsd:string	Any process identifier suitable for diagnostics.
ExtraData	unboundedXMLAnyNamespace	Extra fault data.
Component	xsd:string	Path to component raising the fault.
Stack	stringListType	Optional stack trace.

Implementations SHOULD include a component reference if it is known. Implementations SHOULD also include hostname and process information. Process information may be a low-level identifier (such as an operating system process ID), or it may be some application specific identifier. Its role is merely to distinguish which process amongst many in a load-balanced implementation raised the fault.

4.5.5 Component Faults

Most faults thrown by components are of the type `ComponentFault` which does no extension of the base `DeploymentFault`. The one exception is the fault thrown due to errors in state transition operations.

When a lifecycle state transition action is invoked, the system may fail to properly achieve the next state. As defined by that action, it will generate a `StateActionFault` to indicate failure of the action. In addition to the failure, the fault will indicate the extended state information that would normally be part of a status query directly inline within the fault. The fault may also include other data as defined by the `DeploymentFault`. This fault is defined by the following XML Schema fragment.

```
<xsd:complexType name="StateActionFault">
  <xsd:complexContent>
    <xsd:extension base="DeploymentFaultType">
      <xsd:sequence>
        <xsd:element name="ExtendedState" type="unboundedXMLAnyNamespace"
minOccurs="0"/>
      </xsd:sequence>
    </xsd:extension>
</xsd:complexContent>
</xsd:complexType>
```

4.6 Events

During the course of a deployment, many different lifecycle events will occur. Components are able to notify and receive notifications for these events, as well as forward these events to the Deployment API or other registered listeners. The CDL document can further describe the deployment process by declaring deployment events and event handlers.

Any deployment component in the deployment graph is a candidate for generating and receiving events. If the root of the deployment graph is specified as the recipient, the system EPR will be notified.

4.6.1 Event Format

The implementation of the following events MUST be done using the WS-BaseNotification specification. Additional notification mechanisms MAY also be used without restriction. The component model requires that a component MUST create a notification subscription for each event defined. The WS-TopicSpace for component events defines two topics, lifecycle events and property change events. The TopicSpace and the following events are normatively defined in Appendix A.

As described in the Deployment API, there can be no guarantee of fault tolerant event subscriptions. Implementations of the Component Model MAY choose to include WS-Policy or other metadata to inform components how to renew subscriptions in the event of failures.

4.6.1.1 ComponentLifecycleEvent

The lifecycle event extends from the WSDM `ManagementEventType`. Within the core MUWS event, the resource identifier and endpoint address are accompanied by an event identifier and a timestamp of the event. The CDDLM `ComponentLifecycleEvent` simply augments this base by providing an indication of the lifecycle transition which has occurred.

Events can be derived from this event type as defined by the WSDM MUWS specification, which allows both attributes and elements to be added to the main lifecycle event element.

4.6.1.2 PropertyChangeEvent

When a property of a deployment component changes, it can notify any listeners of the change. The change event is also represented as an extension of the WSDM `ManagementEventType`. The CDDLM `PropertyChangeEvent` adds the WS-ResourceProperties change notification element to support transmission of the old and new element values.

4.6.2 Event Registration

Within the CDL document declaring components, event notifications will also be declared. All event declarations will take the same form. A deployment component will contain a CDL property that declares which event to register a notification for and the target of the notification. The component model requires that a deployment component MUST support the automatic registration of these event handlers at the time of component instantiation.

If it is desired to notify more than one component of the occurrence of a specific event, the property MAY be multiply defined with different target components. All target components MUST be deployment components or component delegates. They MUST NOT be language components or deployment components that are children of component delegates.

In the event of component failures, any restarted components SHOULD ensure that prior event registrations are restored.

[TODO: Must elucidate on how to use WS-ResourceLifetime description of notifications for these events. Also, should ponder on renewable resources.]

[TODO: It would be nice to include other features like onevent set a property, onevent notify some external URI, onevent restart a component, etc…]

4.6.2.1 OnInitialized

When a component transitions to the initialized state, it can notify one or more components that this has occurred. The target attribute will be declared using the CDL reference attribute. This reference is an XPath expression as defined in the CDL Language specification. This expression can be used to find the endpoint reference of the target component, using reference resolution as defined in Section 6.

In addition to notification, the component can also execute a state operation on the remote component. The

```
<xsd:simpleType name="eventProcessEnum">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="initialize"/>
    <xsd:enumeration value="run"/>
    <xsd:enumeration value="terminate"/>
    <xsd:enumeration value="destroy"/>
    <xsd:enumeration value="notify"/>
  </xsd:restriction>
</xsd:simpleType>

<cmp:OnIntialized process=”EventProcessEnum” target=”cdl:ref”>
```

4.6.2.2 OnRunning

When a component has begun the running state, it can notify one or more components that this has occurred.

```
<cmp:OnRunning process=”EventProcessEnum” target=”cdl:ref”>
```

4.6.2.3 OnFailed

When a component has failed, it can notify one or more components that this has occurred.

```
<cmp:OnFailed process=”EventProcessEnum” target=”cdl:ref”>
```

4.6.2.4 OnTerminated

When a component has terminated, it can notify one or more components that this has occurred.

```
<cmp:OnTerminated process=”EventProcessEnum” target=”cdl:ref”>
```

4.6.3 Property Change Notifications

During the lifecycle of a component, it may be necessary to respond to other changes in the deployment, outside of lifecycle events. The WSRF structure allows consumers to register for notifications of changes in the properties of a stateful resource. We will enable this notification to be setup in the CDL document using the following syntax.

```
<cmp:OnChange notify=”cdl:ref”>
```

This will enable a component to be notified of the property change of the element at the cdl:ref destination.

4.6.4 Fault Notifications

A component may also wish to be notified that a fault was thrown by another component or deployment operation. It may subscribe for this notification by using the following syntax in the CDL Language

```
<cmp:OnFault notify=”cdl:ref” faultName=”xs:QName” faultType=”xs:QName”>
```

The fault handling can optionally be scoped by including the fault name or the fault type in the handler’s declaration

4.7 Control Flow

Many deployments follow simple rules, such as deploy these components in order. Or, deploy these components in any order you see fit. In order use a declarative means of creating our deployment description and retaining a simple syntax, we will add a few basic process elements. Basic flow control will allow an implementation to make simple decisions based on the state of deployment components or attributes. We will choose to use a very small subset of standard control primitives to aid in this task

The following sections describe the definition and syntax of these introduced constructs. The syntax borrows heavily from WS-BPEL and other languages for process control. It is not a complete calculus, nor does it need to be. It is minimally defined with only the operations necessary for simple control of deployment. For a discussion of complex deployment scenarios, plus consult Section 5.

When a system is defined, all top level objects are implicit children of a root system node in the graph, which can be equated to the system EPR. When a flow control element is applied, it will be done as a child of a property node or the cdl:system declaration which will make this a child of the root system object. When applied, a flow control element will apply to all children of the node applied to.

As an example, if we wish to have several components deployed in order, we can use the `<cmp:sequence>` element defined below. As this node is declared at the root of the

cdl:system structure, it will encapsulate the other declared elements within its control body.

```
<cdl:cdl
  targetNamespace=”http://example.org/webapp-template”>
<cdl:system>
<cmp:sequence lifecycle=”initialization”>
<WebApplication>
  <cmp:codeBase>http://server/file.jar</cmp:codeBase>
  <cmp:fileName>ApacheDeployer</cmp:fileName>
  <port>80</port>
  <hostname>www.example.org</hostname>
</WebApplication>
<Database>
…
</Database>
<TransactionMonitor>
…
</TransactionMonitor>
</cmp:sequence>
</cdl:system>
</cdl:cdl>
```

All flow control elements depend on lexical order and hierarchy of the declared cdl:system to apply their control functions. A `<cmp:sequence>` element as used above uses the lexical order of the declaration to order the deployment. By default, without the presence of a flow control element, an implementer of CDDLM is free to apply any policy to ordering component operations. Once a flow declaration has been made, it applies to all nodes in the graph that are peers or children of the element in which it is declared, effectively setting a scope of policy. That policy can be changed on more restrictive scopes, such as through redefinition on one of a node’s children, but cannot be redeclared as a child of the same element. An implementer of CDL SHOULD ignore any redeclaration. For a comprehensive discussion of deployment scope, please consult Section 5.1.

4.7.1 Sequence

The sequence operator defines a structure for a deployment operation to be done in sequential order. As elements are defined in the CDL document, their lexical order will be used to define this deployment order. A sequential operation MUST be done as a synchronous, blocking operation. As one element is being initialized, the system MUST wait for initialization to complete before initialization of the next element.

```
<xsd:simpleType name="lifecycleProcessEnum">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="initialization"/>
    <xsd:enumeration value="execution"/>
    <xsd:enumeration value="termination"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:element name=”sequence”>
  <xsd:complexType>
    <xsd:attribute name="lifecycle" type="lifecycleProcessEnum"
use="required"/>
  </xsd:complexType>
</xsd:element>
```

4.7.2 Reverse

The reverse operator defines a structure for a deployment operation to be done in the reverse of sequential order. This element will synchronously execute the deployment operation in the reverse of the declared lexical order.

```
<xsd:element name=”reverse”>
  <xsd:complexType>
    <xsd:attribute name="lifecycle" type="LifecycleProcessEnum"
use="required"/>
  </xsd:complexType>
</xsd:element>
```

4.7.3 Flow

It is often not important what order operations are taken. The flow operator allows the deployment process to be executed concurrently, irrespective of order of declaration. Once all of the elements declared within the scope of the flow have completed their operation, that segment of the operation is considered complete. Thus, the flow operator does not imply that this operation can be done at any time during the deployment, only that its elements do not require ordered, synchronized operation.

```
<xsd:element name=”flow”>
  <xsd:complexType>
    <xsd:attribute name="lifecycle" type="LifecycleProcessEnum"
use="required"/>
  </xsd:complexType>
</xsd:element>
```

4.7.4 Wait

Many times it is important to have some portion of a deployment delay while another continues. The wait operator allows a portion of the deployment process to be paused for a fixed period of time, or until some absolute time. Within the scope of a sequence, it will delay all pending elements of the sequence. Within a flow, it will pause the execution of all elements lexically declared after the wait element.

```
<cmp:wait lifecycle = “cmp:lifecycleProcess”
       ( duration=”xsd:duration” | until=”xsd:dateTime” )/>
```

4.7.5 Switch

The switch operator is designed to be used to control branches of execution. The switch operator allows a choice to be made based on some property of the deployment graph. The condition element must be a valid XPath calculation on declared properties returning a boolean value. The switch operation transfers execution of the deployment process to the node referred in the matching case. It does not allow one component to be executed instead of another. It is ONLY for changing the flow of execution. All component objects declared in a CDL document are presumed MUST be operational.

```
<cmp:switch lifecycle=”cmp:lifecycleProcess”>
       <cmp:case condition=”bool-expr”>+
              <cdl:ref ref=”cdl:ref”/>
       </cmp:case>
       <cmp:otherwise>
              <cdl:ref ref=”cdl:ref”/>
```

```
        </cmp:otherwise>
</cmp:switch>
```

4.8 Component-specific message sets and messages

Any other message can also be processed by the same endpoint. When the extra messages are to be used between component instances, it is prudent to probe for the class of the destination endpoint before sending messages, particularly a long set of messages. It is RECOMMENDED that a component implement the WSDM ManageabilityCharacteristics capability in order to provide this type of discovery. This WSDM capability defines the ManagementCapability resource property.

Note that the notion of *change* is different in an XML based infrastructure than classic compile-time binding. Changes to existing messages may not be significant from a backwards compatibility perspective, if the recipient can still provide the semantics that a caller expects. However, a caller built against a recent version of a component may encounter problems when calling an older version. For this reason, it is critical to add new URIs for message sets whenever the WSDL or supported payload of a message changes in a way that could break systems in such a way.

Components SHOULD NOT change their set of supported interfaces during their life, as this prevents users of a component relying on the results of this call for any period of time.

5 Using the Component Model

Each component object within a deployment will transition through several states during its operation. The CDDLM framework must be able to coordinate those transitions as well as provide an assessment of the system's state through aggregation of the states of the individual components that comprise the system. This section of the document provides comprehensive semantics for defining the state of a deployment, rules for using the component model, and examples for use.

5.1 Scope

The properties contained in a CDL configuration document typically imply very little about the form of a deployment, rather they describe the elements to deploy. As described earlier, by default all components are top level peers of each other. Some of the syntax elements described in the prior chapter can be used to change this behavior and create deployment dependencies which will force order to the deployment.

5.1.1 Delegates

If a component is a delegate, then the deployment structure begins to form as a delegate implies a parent-child relationship among subcomponents. In the diagram below, the independent components, Components A and B, and the component delegate, Component C, exist at the top level of the graph. Components D and E in the diagram are children of Component C.

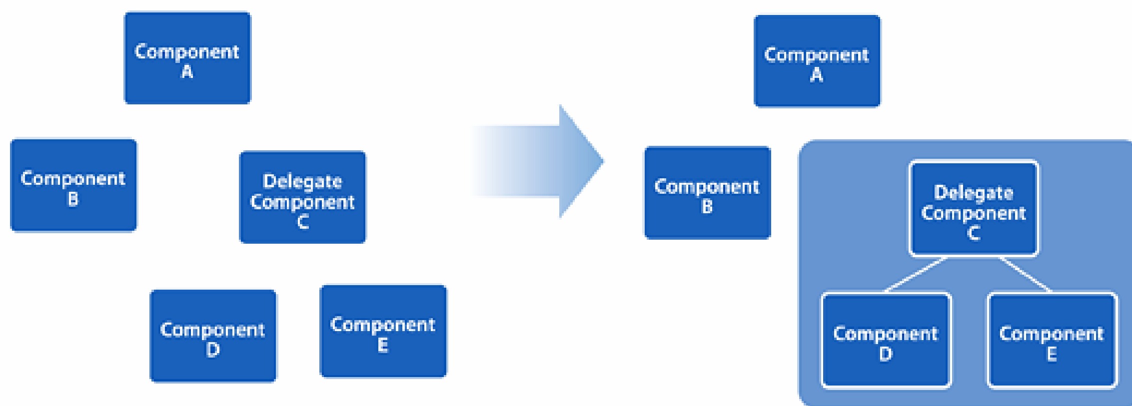

Figure 7. Deployment Scope.

When a component is declared as a delegate, its children are removed from scope of other components not declared as children of the delegate. This restricts further dependencies that are created within the graph to only exist between components in the same relative scopes. If Component B in the diagram had a dependency on Component E, it MUST declare the dependency on Component C and trust that the delegate would enforce the desired behavior. In this sense, Component C becomes responsible for the aggregate lifecycles and dependencies of the objects it is a delegate for.

These dependencies includes lifecycle events and notifications, but not properties or property change notifications. If Component B needed a property value from Component D, it would not have to ask Component C for that value. But, it is an error for Component B to register for a lifecycle event on Component D. Further, if a flow control element were applied to this scenario, such as a sequence, Components A, B, and C would activate sequentially, but this flow element has no bearing on the process used to activate Components D and E. Only Component C has authority to activate those components.

There is no restriction on the depth of delegation allowed in this model. If a component is declared to be a child of a component delegate, it may also be a delegate itself, further responsible for its own children. Each level of delegation provides a more restrictive scope in which to apply the rules of deployment. Implicitly, if Component E in the above diagram were to have its own children F and G, Component D would not be allowed to imply any dependencies upon F or G, only through D itself.

5.1.2 Document Structure

When the CDL document is created, the property lists may declare components all at one level as children of the cdl:system element, or may have some other lexical hierarchy. In the first case, the deployment graph would look like a simple list '(A B C D E), where the second would appear to be a tree or nested list '(A '('(B C) D) E). In the strict syntax of CDL, property lists are trees and would require intermediate nodes to represent as this nested list, this syntax is used solely to show scope in a condensed fashion.

When flow control elements are added to the document, this hierarchy becomes important. In the first example, if a sequence element were declared as a child of the system element, the deployment would proceed as A->B->C->D->E. If the sequence element were declared as a child of one of the other elements, the deployment would happen at random, as the default behavior of the system is to deploy children of the system element as if a flow element were present.

This is due to the scope of application of the flow control element. When a flow control element is used, it applies to the children of the element it is declared upon. In the second example, if a sequence element were declared as a child of element A, then both the sequences A->B->C->D->E and E->A->B->C->D would be valid.

By applying flow control elements further into the tree, successively more restrictive scopes can be created. It is possible to redeclare a flow control element on a child element of a node where one was already declared. In the second example, declaring a flow element as a child of element A instead of a sequence element will not change the random order policy effected by the cdl:system node, but will instead mark a scope of flow control that may be used by other elements. This declaration would also ensure that if this element were imported into another CDL configuration document, it would enforce its desired behavior, rather than be at the whim of the declaration in the new document.

5.2 Synchronization

Most operations of a deployment are done asynchronously. However, several of the properties contained in a CDL configuration document imply the need for synchronicity or wait states. Due to this, the basic hierarchy defined in the document can be modified further to reflect the order required by synchronization.

5.2.1 Events

Using events, one can declare that once Component A is running, it should instruct Component B to begin running. This can allow a deployment to be ordered and walked through its transitions in a simple and declarative manner. One could also instruct Component B that in the event it fails, it should notify Component A of this state transition. Then Component A could invoke some behavior such as a restart of Component B, or termination of itself and/or Component B.

Using this approach, we can create a deployment graph that is not singly linked, such as in a parent-child relationship like delegation. Each deployment action and event can independently link one component to one or more dependent components. If we were to expand the diagram from above to include this event example, it would appear as follows. This diagram is simple, but shows that the evolving graph has a chance to exhibit cyclic behaviors.

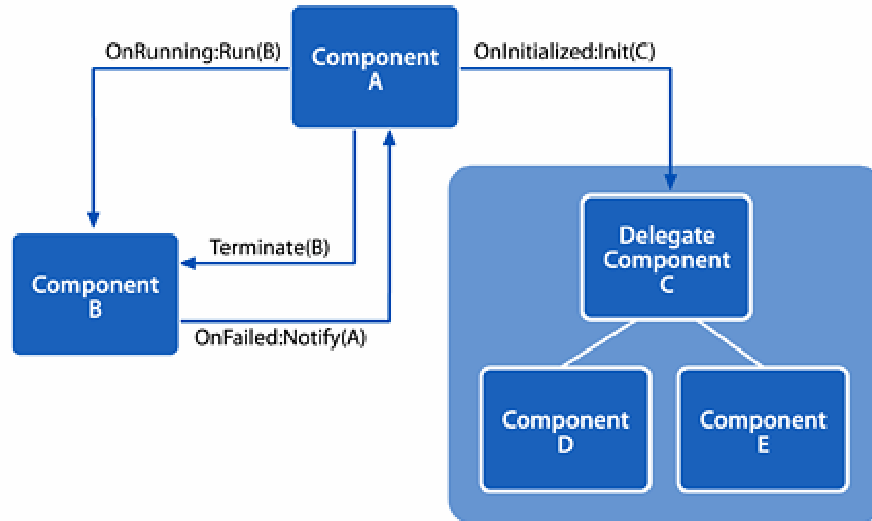

Figure 8. Deployment Event Structure.

The notification event is defined to be performed asynchronously, with no expectation from Component B that Component A will take any specific action. However, it is known, a priori, that once Component A is running, it will inform Component B to run. An implementation of the component model MUST enforce this behavior. This requires that Component B waits for Component A to enter the running state and that no other deployment process tells Component B to execute.

As an example, if Components A, B and C were to be declared as children of a flow element, they would by default initialize concurrently. If Component B were declared to initialize Component C after it has initialized, then Components A and B would be initialized concurrently, but Component C would only be initialized after B has initiated the action.

Applying a sequence control flow element is functionally equivalent to declaring an event handler chain among the processes involved in the sequence. It is not required in an implementation of the component model to realize this structure in that fashion. However, one must recognize that if a sequence operation is declared and a CDL declaration inserts a notification within the sequence, the sequence will still continue in its own manner. If the consumer of the event causes other parts of the graph to change state, the implementer must recognize that state change events may not fire on those portions of the graph at the expected interval. Thus, the implementation of the sequence should be made resilient to this case.

5.2.2 Synchronization Points

It is often not important what order operations are taken. The flow operator allows the deployment process to be executed independent of its lexical declaration. A flow operator, as all other flow control elements, implies a synchronization point at the end of its scope. If a node has children that are declared with the flow operator, the node will

still synchronize to its peers if necessary even if its children are executing concurrently. Thus, the completion of a deployment command is a guarantee of synchronization.

As an example, a CDL document may declare a deployment as follows (note that most of the declaration has been omitted for simplicity):

```
<cdl:cdl
  targetNamespace="http://example.org/sync-template">
<cdl:system>
<cmp:sequence lifecycle="initialization">
<ComponentA/>
<ComponentB>
       <cmp:flow lifecycle="initialization">
       <Component D/>
       <Component E/>
</ComponentB>
<ComponentC/>
</cmp:sequence>
</cdl:system>
</cdl:cdl>
```

In this case, the sequence operator defines that all children of the cdl:system node should be executed in order. This operator is overridden by the flow operator applied to Component B. This creates a scope encompassing Components' B, D and E. It implies that Component A must initialize before B before C. Components B, D and E can be initialized in any order, but only after Component A has completed its initialization. Finally, Component C can only initialize after Components B, D and E.

If instead, the subscope of Component B were modified with a sequence operator, Component B would not be considered initialized until after Component E had completed its initialization. This defines a synchronization point at the scope of the flow control operator rather than at the component itself. Similarly, the switch and wait operators create synchronization points at the node which they are declared.

5.3 Aggregate System State

Though the aggregate state of the deployment components is reflected in the system state, each component within the system is NOT REQUIRED to exist within the same state. Deployments may be constantly in flux, with components failing during the operations of the system. The system MAY respond to the failure(s) without affecting the state of the whole, but the following recommendations should be followed. If all components are in a particular state, the system MUST report its state similarly. If the system is transitioning to a state, such as from instantiated to initialized, the system SHOULD report its state as the prior state, instantiated, until all components have reached the new state, initialized.

The system MUST also invoke actions in a coordinated fashion for all components. If two components are defined in a system then both must be initialized before they are run and so on. If the objects fail to transition to the same states, then the framework SHOULD coordinate their termination. Thus, if an object fails to initialize, the system SHOULD report its state as failed as well. The CDDLM framework MAY, however, choose to reset and resume deployment for the failed portion of the task, or fail the entire

deployment phase. It is important to note that the specific state of the system can only be determined by querying the system EPR. Any use of component states by entities outside of the CDDLM framework or the system EPR is undefined.

If deployment event notifications are declared such as OnFailed events, the framework MAY wait to observe the results of the event propagation. If the events are declared in the CDL document as process steps rather than as events, such as <OnFailed process="terminate">, the framework SHOULD wait for the actions to propagate.

CDDLM also enables component delegates to aggregate the lifecycle of other components. Using this aggregation, the main component represents a single instance of lifecycle state to the system, the sub components being opaque to the system in this regard. A component delegate MAY manage transitions of its children in any way it sees fit. For example, a delegate could choose to automatically restart a failed child node, but not need to indicate the failure to the system. The delegate itself, though, MUST follow the semantics of a deployment component itself with respect to the rest of the system. In this way, a delegate can be used to encode complex rules or interdependencies between components but still enforce the basic semantics of the component model.

5.4 Controlling Deployments

With effective use of just events and delegation, many complex deployment scenarios could be accommodated. However, the goal of the component model is to enable loosely coupled, scalable patterns for deployment. Events are an important tool for deployment, but may lead to significant extra effort to manage and maintain. For example, if we wanted to deploy our example system by deploying some components A then B then C, we would need to declare two events. That is, one less than the number of components to deploy. If we had a deployment with hundreds of components, the event system would be unwieldy at best.

Using delegation, one could create a complete structure for a deployment. However, in a large deployment, this will tightly couple components' behavior. Further, much of the rules describing how to react to deployment events will be lost within the components themselves. By creating a chain of parents and children, there is a potential for the relationships to restrict the form of the deployment and cause unwanted side effects. It is recommended to use delegation sparingly. There are many cases, though, in which the delegation model will help with deployment.

5.4.1 Event Handlers

In order to handle more complex deployment scenarios, components can be defined whose sole purpose is to handle events. This allows a component to take responsibility for handling an event and determining itself how to respond to state transitions. These components should either be defined outside of the scope of a deployment process flow, or have simple internal state transitions of its own. These globally scoped event handlers can then be used as utilities to handle more complex operations such as restart on failure, or immediate termination on failure.

As an example, a component may define a notification to be sent to another component in the event that it fails. This component will remain in the failed state until the event handler chooses its course of action. The event handler MAY choose to perform an internal action to reset or fix the resource, in which case the component MUST transition back to the state it was in prior to failing. Otherwise, the event handler should perform its actions and forward the event to the system.

6 Reference Resolution

Many properties in the CDL Language use the cdl:ref attribute to define their target location. Once a component is deployed, this reference must be translated into an actual resource identifier. A component reference will translate into a component EPR. A property reference will translate into its component owner's EPR and the attribute resource name under that EPR.

If a component property is set to a CDL reference, that property should be flagged by the component. Before the property is used, the component MUST retrieve the value of the property. Currently, the only available means to perform this translation is for a component to execute the `resolve`() function of the system EPR as defined in the Deployment API. Once a component retrieves the EPR of the destination, it can make a request to that EPR for the resource property of the target endpoint.

Once the property value is retrieved, it MUST be substituted for the reference. The component also MUST block whatever step it is about to perform and wait for the reference resolution process. It is RECOMMENDED that a component scan its property list before taking any deployment action that would require the use of the property. If it is unknown when the property is used, the component SHOULD retrieve this value before the component is initialized.

This process requires that the system EPR is able to handle the scale of potential resolution requests. Also, each component must have some way to discover the system EPR. We have not defined these mechanisms here.

7 Security Considerations

For security, the CDDLM Component Model relies on the security provided by the CDL specification for securing the description of a deployment. Web Services security mechanisms, including transport-level security and SOAP Message Security are presumed to be used in order to secure the connections between components and EPRs.

The components themselves are responsible for performing actions within a trusted host. It is important that proper trust is established between host nodes in the deployment. Operations may require elevated permissions in order to perform. The component nodes MUST ensure that those rights cannot be usurped or delegated. The maintenance action

task listed is most susceptible to security issues, as it may be possible to use this command to run arbitrary programs on remote nodes.

8 Implementation Notes

In translating these design concepts into a concrete model, there are several goals and guiding principles that are used. It is the desire of the CDDLM model to be simple, flexible, easy to implement, but provide a rich set of behaviors that are capable of even the most complex deployments.

8.1.1 Loose Coupling

Performing any task at Grid scale is difficult. In a deployment on a virtually unlimited number of nodes, there will be an extremely large set of operations, state changes and events occurring simultaneously and possibly asynchronously. This can potentially lead to ripples of side effects that can cause one component failure to bring down an entire deployed system or multiple systems. Additionally, in a large deployment any one change or maintenance operation on one component or node can have effect on the entire system.

Loose coupling is taken a meaning that software elements in a system are only very lazily joined to other parts in the system, so being resilient to changes in the implementation. Tight coupling is generally acknowledge as harmful in a distributed system, as it makes it hard to upgrade software elements in isolation. The Distributed Objects pattern (ANSA, CORBA, DCOM, RMI, EJB) has a reputation for excessively tight coupling. COM and CORBA require that the signature of an interface remain unchanged, while Java RMI allows new methods on an interface to be added, but requires existing ones to be retained. All of these systems require compile-time binding, creating a tight link between implementation and use.

The CDDLM Component Model is built on the WS Resource Framework and other WS-I defined standards. The WSRF presents some risk of enabling tight coupling. We have employed several tactics to minimize that risk.

8.1.2 Declarative Configuration rather than Direct Invocation

The primary way for one component to configure and use a third party component should be pre-deployment configuration of the component via the Deployment API. That is, instead, of one component sending *set*() messages to another component, it should invoke the Deployment API and request that attributes of that component get set. When the remote component is deployed, it will read and use those attributes which it needs.

This is a lesson from Apache Ant [9]; the XML-level configuration has proved far more robust than the software API used between tasks. It is the risk that another task has used one of the attribute setters that prevents the development team from removing setter methods that are no longer used at the XML level. For example, a method setDest (String) may be replaced by the method setDest(File), the runtime automatically mapping the XML string to a project-relative File instance. Even with the new method used by the

runtime, the old method needs to be retained purely for software that invoked the class directly. Nor can anything that uses the new method call an old version of the Task; that will raise an exception.

By having components set attributes on the deployment graph, rather than invoke the remote component directly, we can decouple the components more. There is still the problem of keeping the types of attributes consistent; with a limited set of types or a loose type system this ought to be relatively straightforward.

8.1.3 Minimal use of Distributed Object concepts

The CDDLM Component Model is not a distributed object system. Thus, there are no requirements that it implement Distributed Object patterns: factories, callbacks, leased lifetimes. We will use the minimum subset of WSRF option as we need – no more, no less. Additionally, this model will avoid exporting the interfaces of the underlying object instance. There is no declared native interface such as Java or C++ objects. This will allow us to avoid requiring systems to implement the same underlying object systems or implementations.

An application developer choosing to develop to this model is free to make several choices. A specific component object does not need to be an object within some other object model such as J2EE, COM or .NET. It is allowed to be an executable application, or a component within a host. The only requirement is that the attribute assignment properties of the CDL language must be able to indicate the proper means of activation of the component object. As an example, the SmartFrog implementation chooses to map a language component to a SmartFrog class (which is the deployment component) through the sfClass attribute. The current component model includes basic bindings for SmartFrog, Java, Ant, and .NET components. Others can be created by extending the component model schema.

In the case of inherited behavior, a derived component may execute the methods of its parent or override its behavior completely. It is up to the implementer to choose and determine the methodology to be used. This is to say that the CDDLM Component Model imposes no constraints on component object inheritance.

8.1.4 Do not rely on schema validation

XML schemas that are provided can be used to ensure that valid documents are created. The schemas are a hint. There is no guarantee that a sender or recipient will produce compliant documents. If operations are unsuccessful for failure to pass schema validation, many unwanted problems may occur.

8.1.5 Pass data in messages, not as references to resources

A common pattern in in-process objects is for those objects to pass references to themselves or other objects around; the recipient calls getter and setter methods to complete the operation. This is a good information-hiding practice for local objects, but excessively chatty for remote connections. It also vastly increases the requirements of a

caller of the service. Instead of being able to call a service with a message it constructs, it must instead provide an accessible endpoint to call back to, thus binding the two remote objects for the duration of the operation. Over time, the two objects end up holding many remote references to each other causing them to be excessively coupled.

8.1.6 Make effective use of the proposed `ImplementsMessageSet` message

The proposed `ImplementsMessageSet` message is intended to probe an endpoint to see if it has implemented a specific set of messages, similar to the `instanceof` test for interfaces. However, the Java language test merely verifies that the class implements an interface of a given name; it makes no guarantee that the interface behaves as expected.

The original introduction of the interface concept was by David .L. Parnas in his 1974 paper on information hiding [10]. In this paper, Parnas defined an interface as both signature *and* semantics. The `interfaceof` test only verifies the signature -for robust coupling we need to ensure that the semantics are consistent.

Only if an endpoint processes a set of messages in compliance with the specification of that interface, can it declare itself as implementing that particular message set. If a feature change means that the processing of those messages has changed such that they no longer match the interface specification, then the endpoint must not declare that it implements the interface.

This strongly resembles COM's QueryInterface() design in intent, although the general implementation of QueryInterface() fell somewhat short of the goal, as it has essentially devolved to declaring nothing but signature compatibility. Proof of this is the E_NOTIMPL return code, which allows any method in an interface to return saying "actually, we do not do anything".

Of these proposed tactics, the use of attributes for declarative coupling is the most CDDLM-specific, and promises the loosest coupling. The rules for processing our proposed `ImplementsMessageSet` message are similar to the ideals of COM, but not its reality. We should strive to do better.

9 Editor Information

Stuart Schaefer
Softricity, Inc.
27 Melcher Street
Boston, MA 02210
Email: ssschaefer@softricity.com

Steve Loughran
Internet Systems and Storage Laboratory
Hewlett-Packard Laboratories
Filton Road
Stoke Gifford
Bristol BS34 8QZ
United Kingdom
Email: steve_loughran@hpl.hp.com

10 Acknowledgements

The authors of this document would like to acknowledge the contributions, assistance and support of the following people and their respective companies: Dejan Milojicic, Takashi Kojo, Jun Tatemura, Peter Toft, Julio Guijarro and Vanish Talwar.

References

1. Foster, I., Kesselman, C., Nick, J. and Tuecke, S. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Globus Project, 2002. www.globus.org/research/papers/ogsa.pdf.
2. Foster, I., Kesselman, C. and Tuecke, S. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, *15* (3). 200-222. 2001.
3. Tuecke, S., Czajkowski, K., Foster, I., Frey, J., Graham, S. and Kesselman, C. Grid Service Specification, 2002. www.globus.org/research/papers/gsspec.pdf.
4. Loughran, S. *WS-RF and CDDLM* , 2004
5. Tatemura, J. XML Configuration Description Language Specification , 2004.
6. Loughran, S. Deployment API Draft, 2004.
7. Tuecke, S. et al., Web Services Base Faults WS-BaseFaults
8. Graham, S. et al., Web Services Resource Properties (WS-ResourceProperties)
9. Duncan-Davidson, J., *Apache Ant*.
10. Parnas, D. L. *On the Criteria to Be Used in Decomposing Systems into Modules*, Communications of the ACM, Vol. 15, No. 12, pp.1053-1058, 1972.
11. Bradner, S. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119.
12. Loughran, S. Making Web Services that Work, HP Laboratories, TR-HPL-2002-274, 2002.

Appendix A – Component Object Definitions

A.1 XML Schema

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns="http://www.gridforum.org/cddlm/components/2005/02"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:cdl="http://www.gridforum.org/2004/12/CDDLM/XML-CDL/1.0"
xmlns:wsp="http://schemas.xmlsoap.org/ws/2002/12/policy"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
xmlns:wsrp="http://www.ibm.com/xmlns/stdwip/web-services/WS-BaseFaults"
xmlns:ns1="http://www.ibm.com/xmlns/stdwip/web-services/WSResourceProperties"
xmlns:wsrl="http://www.ibm.com/xmlns/stdwip/web-services/WSResourceLifetime"
xmlns:wsnt="http://www.ibm.com/xmlns/stdwip/web-services/WSBaseNotification"
xmlns:wstop="http://www.ibm.com/xmlns/stdwip/web-services/WSTopics" xmlns:muws-
xs="http://docs.oasis-open.org/wsdm/2004/07/muws-1.0/schema" xmlns:mows-
xs="http://docs.oasis-open.org/wsdm/2004/07/mows-1.0/schema"
targetNamespace="http://www.gridforum.org/cddlm/components/2005/02"
elementFormDefault="qualified" id="components">
  <!-- ============================================================ -->
  <xs:annotation>
    <xs:documentation>

   This is the XSD describing the types of the CDDLM Component Model


   Version: 1.0

   The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL
   NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED",  "MAY", and
   "OPTIONAL" in this document are to be interpreted as described in
   RFC 2119.
   http://www.ietf.org/rfc/rfc2119.txt

    </xs:documentation>
  </xs:annotation>
  <!-- ============================================================ -->
  <!--IMPORTS -->
  <!-- ============================================================ -->
  <xs:import namespace="http://www.gridforum.org/2004/12/CDDLM/XML-CDL/1.0"
schemaLocation="xml-cdl.xsd"/>
  <xs:import namespace="http://schemas.xmlsoap.org/ws/2003/03/addressing"
schemaLocation="wsrf/ws-addressing.xsd"/>
  <xs:import namespace="http://www.ibm.com/xmlns/stdwip/web-services/WS-
BaseFaults" schemaLocation="wsrf/WS-BaseFaults.xsd"/>
  <!-- ============================================================ -->
  <!--PROPERTIES -->
  <!-- ============================================================ -->
  <xs:simpleType name="lifecycleStateEnum">
    <xs:annotation>
      <xs:documentation>Enumeration of the valid states of a deployment
component.</xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
      <xs:enumeration value="undefined">
        <xs:annotation>
          <xs:documentation>Undefined state.</xs:documentation>
        </xs:annotation>
      </xs:enumeration>
      <xs:enumeration value="instantiated">
        <xs:annotation>
          <xs:documentation>Instantiated system</xs:documentation>
        </xs:annotation>
      </xs:enumeration>
      <xs:enumeration value="initialized">
```

```
        <xs:annotation>
          <xs:documentation>Initialized system</xs:documentation>
        </xs:annotation>
      </xs:enumeration>
      <xs:enumeration value="running">
        <xs:annotation>
          <xs:documentation>Running system</xs:documentation>
        </xs:annotation>
      </xs:enumeration>
      <xs:enumeration value="failed">
        <xs:annotation>
          <xs:documentation>System has failed</xs:documentation>
        </xs:annotation>
      </xs:enumeration>
      <xs:enumeration value="terminated">
        <xs:annotation>
          <xs:documentation>System has terminated</xs:documentation>
        </xs:annotation>
      </xs:enumeration>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="lifecycleProcessEnum">
    <xs:restriction base="xs:string">
      <xs:enumeration value="initialization">
        <xs:annotation>
          <xs:documentation>Apply the process during the intialization phase of
the deployment.</xs:documentation>
        </xs:annotation>
      </xs:enumeration>
      <xs:enumeration value="execution">
        <xs:annotation>
          <xs:documentation>Apply the process during the transition to the
execution phase.</xs:documentation>
        </xs:annotation>
      </xs:enumeration>
      <xs:enumeration value="termination">
        <xs:annotation>
          <xs:documentation>Apply the process during the termination phase of
the deployment.</xs:documentation>
        </xs:annotation>
      </xs:enumeration>
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="terminationRecordType">
    <xs:annotation>
      <xs:documentation>
     Termination Record
     </xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="timestamp" type="xs:dateTime"/>
      <xs:element name="message" type="xs:string" minOccurs="0"/>
      <xs:element name="extraData">
        <xs:complexType>
          <xs:sequence>
            <xs:any namespace="##other" processContents="lax" minOccurs="0"
maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="commandPathType">
    <xs:annotation>
      <xs:documentation>A command path can contain a command and one or more
arguments.</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="args" type="xs:anyType" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
```

```
  <xs:complexType name="componentStatusType">
    <xs:annotation>
      <xs:documentation>This is the status of a deployed
component.</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="state" type="lifecycleStateEnum"/>
      <xs:element name="stateInfo" type="xs:string"/>
      <xs:element name="extendedState" type="xs:anyType" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <!-- ============================================================ -->
  <!-- COMPONENT ELEMENTS  -->
  <!-- ============================================================ -->
  <xs:element name="ComponentReference" type="wsa:EndpointReferenceType">
    <xs:annotation>
      <xs:documentation>Every component will be accessible via its WS-RF
EPR.</xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:element name="CodeBase" type="xs:anyURI">
    <xs:annotation>
      <xs:documentation>Location of the component file
assets.</xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:element name="CommandPath" type="commandPathType">
    <xs:annotation>
      <xs:documentation>The location or object within the assets of the
component to use in activating the component.</xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:element name="Delegate" type="xs:string" nillable="true">
    <xs:annotation>
      <xs:documentation>A component which controls other
components.</xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:element name="DeploymentProperties">
    <xs:annotation>
      <xs:documentation>The WSRF Resource Property document fragment of a
component.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:choice>
        <xs:element name="propertyDocumentRef" type="xs:anyURI"/>
        <xs:element name="propertyDocumentFragment" type="xs:anyType"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
  <!-- ============================================================ -->
  <!-- DEPLOYMENT OPERATORS -->
  <!-- ============================================================ -->
  <xs:element name="sequence">
    <xs:annotation>
      <xs:documentation>A sequential deployment operator.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:attribute name="lifecycle" type="lifecycleProcessEnum"
use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="flow">
    <xs:annotation>
      <xs:documentation>A deployment operator for creating concurrent
deployment processes.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:attribute name="lifecycle" type="lifecycleProcessEnum"
use="required"/>
    </xs:complexType>
  </xs:element>
```

```
  <xs:element name="wait">
    <xs:annotation>
      <xs:documentation>A deployment operator for blocking a deployment process
for a period of time.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:attribute name="duration" type="xs:duration"/>
      <xs:attribute name="until" type="xs:dateTime"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="switch">
    <xs:annotation>
      <xs:documentation>A deployment operator for branching deployment
processes.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="case" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="otherwise"/>
      </xs:sequence>
      <xs:attribute name="lifecycle" type="lifecycleProcessEnum"
use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="case">
    <xs:annotation>
      <xs:documentation>A branch of a deployment process.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:all>
        <xs:element ref="cdl:ref"/>
      </xs:all>
      <xs:attribute name="condition" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="otherwise">
    <xs:annotation>
      <xs:documentation>A default branch of a deployment
process.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:all>
        <xs:element ref="cdl:ref"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
  <!-- ============================================================ -->
  <!-- DEPLOYMENT EVENTS  -->
  <!-- ============================================================ -->
  <xs:element name="OnInitialized">
    <xs:annotation>
      <xs:documentation>Notify a component that the source has achieved the
initialized state.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:attribute name="notify" type="cdl:pathType" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="OnRunning">
    <xs:annotation>
      <xs:documentation>Notify a component that the source has achieved the
running state.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:attribute name="notify" type="cdl:pathType" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="OnFailed">
    <xs:annotation>
      <xs:documentation>Notify a component that the source has entered the
failed state.</xs:documentation>
    </xs:annotation>
```

```
    <xs:complexType>
      <xs:attribute name="notify" type="cdl:pathType" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="OnTerminated">
    <xs:annotation>
      <xs:documentation>Notify a component that the source has entered the
terminated state.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:attribute name="notify" type="cdl:pathType" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="OnChange">
    <xs:annotation>
      <xs:documentation>Notify a component that a property of the source has
changed.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:attribute name="notify" type="cdl:pathType" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="OnFault">
    <xs:annotation>
      <xs:documentation>Subscribe to a component to be notified of the
occurence of a named fault.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:attribute name="ref" type="cdl:pathType" use="required"/>
      <xs:attribute name="faultName" type="xs:QName" use="optional"/>
      <xs:attribute name="faultType" type="xs:QName" use="optional"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

A.2 WSDL 1.1

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:cmp="http://www.gridforum.org/cddlm/components/2005/02"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing" xmlns:wsrf-
bf="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-BaseFaults-1.2-draft-
01.xsd" xmlns:wsrf-rp="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceProperties-1.2-draft-01.xsd" xmlns:wsrf-rl="http://docs.oasis-
open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-1.2-draft-01.xsd" xmlns:wsrf-
nt="http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-1.2-draft-
01.xsd" xmlns:muws-p1-xs="http://docs.oasis-open.org/wsdm/2004/12/muws/wsdm-
muws-part1.xsd"
targetNamespace="http://www.gridforum.org/cddlm/components/2005/02">
  <!-- ============================================================== -->
  <wsdl:documentation>

   This is the WSDL Describing the Component Model API for CDDLM
   deployment components.

   The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL
   NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED",  "MAY", and
   "OPTIONAL" in this document are to be interpreted as described in
   RFC 2119.
   http://www.ietf.org/rfc/rfc2119.txt

 </wsdl:documentation>
  <!-- ============================================================== -->
  <!-- BEGIN IMPORTS -->
  <!-- ============================================================== -->
```

```
  <wsdl:import namespace="http://www.gridforum.org/cddlm/components/2005/02"
location="component-model.xsd"/>
  <wsdl:import namespace="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceProperties-1.2-draft-01.xsd" location="../wsrf/wsrf-WS-
ResourceProperties-1.2-draft-01.xsd"/>
  <wsdl:import namespace="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceLifetime-1.2-draft-01.xsd" location="../wsrf/wsrf-WS-ResourceLifetime-
1.2-draft-01.xsd"/>
  <!-- ============================================================ -->
  <!-- COMPONENT MESSAGES -->
  <!-- ============================================================ -->
<wsdl:message name="initializeRequest">
   <wsdl:part element="cmp:initializeRequest" name="initializeRequest"/>
 </wsdl:message>
 <wsdl:message name="initializeResponse">
   <wsdl:part element="cmp:initializeResponse" name="initializeResponse"/>
 </wsdl:message>

 <wsdl:message name="runRequest">
   <wsdl:part element="cmp:runRequest" name="runRequest"/>
 </wsdl:message>
 <wsdl:message name="runResponse">
   <wsdl:part element="cmp:runResponse" name="runResponse"/>
 </wsdl:message>

 <wsdl:message name="terminateRequest">
   <wsdl:part element="cmp:terminateRequest" name="terminateRequest"/>
 </wsdl:message>
 <wsdl:message name="terminateResponse">
   <wsdl:part element="cmp:terminateResponse" name="terminateResponse"/>
 </wsdl:message>

 <wsdl:message name="taskActionRequest">
   <wsdl:part element="cmp:taskActionRequest" name="taskActionRequest"/>
 </wsdl:message>
 <wsdl:message name="taskActionResponse">
   <wsdl:part element="cmp:taskActionRequest" name="taskActionRequest"/>
 </wsdl:message>

  <!-- ============================================================ -->
  <!-- COMPONENT ENDPOINT -->
  <!-- ============================================================ -->
  <wsdl:portType name="ComponentEndpoint" wsrf-
rp:resourceProperties="cmp:ComponentProperties">
   <!-- WSRF Property Messages-->
   <wsdl:operation name="GetResourceProperty">
     <wsdl:input name="GetResourcePropertyRequest"
       message="wsrf-rp:GetResourcePropertyRequest"/>
     <wsdl:output name="GetResourcePropertyResponse"
       message="wsrf-rp:GetResourcePropertyResponse"/>
     <wsdl:fault name="ResourceUnknownFault"
       message="wsrf-rp:ResourceUnknownFault"/>
     <wsdl:fault name="InvalidResourcePropertyQNameFault"
       message="wsrf-rp:InvalidResourcePropertyQNameFault"/>
   </wsdl:operation>

   <wsdl:operation name="GetMultipleResourceProperties">
     <wsdl:input name="GetMultipleResourcePropertiesRequest"
       message="wsrf-rp:GetMultipleResourcePropertiesRequest"/>
     <wsdl:output name="GetMultipleResourcePropertiesResponse"
       message="wsrf-rp:GetMultipleResourcePropertiesResponse"/>
     <wsdl:fault name="ResourceUnknownFault"
       message="wsrf-rp:ResourceUnknownFault"/>
     <wsdl:fault name="InvalidResourcePropertyQNameFault"
       message="wsrf-rp:InvalidResourcePropertyQNameFault"/>
   </wsdl:operation>

  <wsdl:operation name="SetResourceProperties">
    <wsdl:input name="SetResourcePropertiesRequest"
    ="wsrf-rp:SetResourcePropertiesRequest"/>
    <wsdl:output name="SetResourcePropertiesResponse"
    ="wsrf-rp:SetResourcePropertiesResponse"/>
```

```
    <wsdl:fault name="ResourceUnknownFault" message="wsrf-
rp:ResourceUnknownFault"/>
    <wsdl:fault name="InvalidSetResourcePropertiesRequestContentFault"
    ="wsrf-rp:InvalidSetResourcePropertiesRequestContentFault"/>
    <wsdl:fault name="UnableToModifyResourcePropertyFault"
    ="wsrf-rp:UnableToModifyResourcePropertyFault"/>
    <wsdl:fault name="InvalidResourcePropertyQNameFault"
    ="wsrf-rp:InvalidResourcePropertyQNameFault"/>
    <wsdl:fault name="SetResourcePropertyRequestFailedFault"
    ="wsrf-rp:SetResourcePropertyRequestFailedFault"/>
  </wsdl:operation>

  <wsdl:operation name="QueryResourceProperties">
    <wsdl:input name="QueryResourcePropertiesRequest"
    ="wsrf-rp:QueryResourcePropertiesRequest"/>
    <wsdl:output name="QueryResourcePropertiesResponse"
    ="wsrf-rp:QueryResourcePropertiesResponse"/>
    <wsdl:fault name="ResourceUnknownFault" message="wsrf-
rp:ResourceUnknownFault"/>
    <wsdl:fault name="InvalidResourcePropertyQNameFault"
    ="wsrf-rp:InvalidResourcePropertyQNameFault"/>
    <wsdl:fault name="UnknownQueryExpressionDialectFault"
    ="wsrf-rp:UnknownQueryExpressionDialectFault"/>
    <wsdl:fault name="InvalidQueryExpressionFault" message="wsrf-
rp:InvalidQueryExpressionFault"/>
    <wsdl:fault name="QueryEvaluationErrorFault" message="wsrf-
rp:QueryEvaluationErrorFault"/>
  </wsdl:operation>

  <!-- WS-RF Resource Lifetime: ImmediateResourceTermination -->
   <wsdl:operation name="Destroy">
     <wsdl:input name="DestroyRequest" message="wsrf-rl:DestroyRequest"/>
     <wsdl:output name="DestroyResponse" message="wsrf-rl:DestroyResponse"/>
     <wsdl:fault name="ResourceUnknownFault" message="wsrf-
rl:ResourceUnknownFault"/>
     <wsdl:fault name="ResourceNotDestroyedFault" message="wsrf-
rl:ResourceNotDestroyedFault"/>
   </wsdl:operation>

  <!-- initialize a component -->
   <wsdl:operation name="Initialize">
     <wsdl:input message="cmp:initializeRequest" name="initializeRequest"/>
     <wsdl:output message="cmp:initializeResponse" name="initializeResponse"/>
     <wsdl:fault name="StateActionFault" message="cmp:StateActionFault"/>
   </wsdl:operation>
  <!-- run a component -->
   <wsdl:operation name="Run">
     <wsdl:input message="cmp:runRequest" name="runRequest"/>
     <wsdl:output message="cmp:runResponse" name="runResponse"/>
     <wsdl:fault name="StateActiontFault" message="cmp:StateActionFault"/>
   </wsdl:operation>
   <!-- terminate a component -->
   <wsdl:operation name="Terminate">
     <wsdl:input message="cmp:terminateRequest" name="terminateRequest"/>
     <wsdl:output message="cmp:terminateResponse" name="terminateResponse"/>
     <wsdl:fault name="StateActionFault" message="cmp:StateActionFault"/>
   </wsdl:operation>

   <!-- maintenance operation -->
   <wsdl:operation name="RunTask">
     <wsdl:input message="cmp:taskActionRequest" name="taskActionRequest"/>
     <wsdl:output message="cmp:taskActionResponsne" name="taskActionResponse"/>
   </wsdl:operation>

  </wsdl:portType>
</wsdl:definitions>
```

A.4 Topic Space

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<wstop:TopicSpace name="LifecycleEvents"
targetNamespace="http://www.gridforum.org/cddlm/components/events/2005/02/event
s" xmlns:cmp="http://www.gridforum.org/cddlm/components/2005/02"
xmlns:wstop="http://docs.oasis-open.org/wsn/2004/06/wsn-WS-Topics-1.2-draft-
01.xsd">

 <wstop:Topic name="LifecycleEvent" messageTypes="cmp:LifecycleEvent"/>
 <wstop:Topic name="PropertyChange" messageTypes="cmp:PropertyChangeEvent"/>
</wstop:TopicSpace>
```