Deployment and Configuration of Component-based Distributed Applications Specification

June 2003
Draft Adopted Specification
ptc/03-07-02

An Adopted Specification of the Object Management Group, Inc.

Contents

Preface

About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 600 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

Specification Overview

The purpose of the Deployment and Configuration (D&C) specification is to define the mechanisms by which *component-based distributed* applications are deployed. Deployment is defined as the processes between acquisition of software and execution of software. We define the software deployer as the agent that acquires software, and performs the activities that prepare for, and possibly perform the eventual execution of the software. This requires specifications for:

- Describing the deployment requirements of the software.
- Packaging the software and associated metadata for delivery between the software producer and the deployer.
- Receiving and configuring the software into the deployer's environment *before* deployment decisions are made.
- Describing the facilities of the targeted distributed execution infrastructure.
- Planning (making decisions for) how the software will be deployed onto the targeted distributed execution infrastructure.
- Performing the actual preparation of the application for execution, e.g., moving parts of the software to their location of execution.
- Launching, monitoring, and terminating the application.

This specification defines a Platform Independent Model (PIM) to address the above issues, which introduces a conceptual basis for deployment systems independent of technology platform. This specification also defines a Platform Specific Model for deployment and configuration for the CORBA Component Model (CCM). Although a PSM was considered for the Software Communication Architecture (SCA), it is not in this specification due to its dependencies on other standards efforts that are in progress. It will be addressed when these efforts can be synchronized.

The scope of this specification defines information models (and implied formats), interfaces and associated semantics for the basic machinery of deployment to enable deployment tools to be written against a standard infrastructure. This will enable tools with varying capabilities, from multiple vendors, to be written and supplied separately from the implementers of the runtime infrastructure for deployment and execution. This specification further defines interfaces between elements of the infrastructure to enable interoperable implementations of parts of a deployment infrastructure.

Design Rationale

This specification defines models sufficient to define an interchange between creators of component-based distributed software, and deployers of that software. Furthermore, this specification provides models to enable a variety of tools to be written to interoperate with different deployment infrastructures. The deployment interfaces based on the models represent a set of building blocks for tools rather than a single model for how such tools should operate to their users.

The scope of this specification defines interfaces to the distributed infrastructure as a whole, and also defines infrastructure interfaces internal to the distributed infrastructure to enable interoperable implementations between the whole/centralized deployment system and the part of that system that runs on specific target computers.

OMG Documents

The OMG Specifications Catalog is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

The OMG documentation is organized as follows:

OMG Modeling Specifications

Includes the UML, MOF, XMI, and CWM specifications.

OMG Middleware Specifications

Includes CORBA/IIOP, IDL/Language Mappings, Specialized CORBA specifications, and CORBA Component Model (CCM).

Platform Specific Model and Interface Specifications

Includes CORBAservices, CORBAfacilities, OMG Domain specifications, OMG Embedded Intelligence specifications, and OMG Security specifications.

Obtaining OMG Documents

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

OMG formal documents are available from our web site in PostScript and PDF format. Contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Helvetica bold - OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier bold` - Programming language elements.

Helvetica - Exceptions

Acknowledgments

The following companies submitted and/or supported this specification:

- 88solutions Corporation
- Carleton University
- Deutsche Telekom
- France Telekom
- Fraunhofer FOKUS
- Humboldt Universität Berlin
- Laboratoire d'Informatique Fondamentale de Lille
- Mercury Computer Systems
- MITRE Corporation

Introduction 1

Contents

This chapter includes the following topics.

"A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. Larger pieces of a system's functionality may be assembled by reusing components as parts in an encompassing component or assembly of components, and wiring together their required and provided ports." [UML2]

In short, the idea of component-based development is to divide an application into small reusable components that can be connected to other components via ports, or, speaking the other way around, to compose applications by reusing and interconnecting existing components. An important idea is recursion, that an assembly — a set of interconnected components — can be seen as a component in itself, and therefore be reused the same way: an assembly always “implements” a specific component interface. Within an assembly, connections must be made between its subcomponents, and arrangements must be made for the assembly's external ports — ports of the component interface that the assembly is implementing — to delegate their behavior to subcomponent ports.

In order to instantiate, or deploy, a component-based application, instances of each subcomponent must first be created, then interconnected and configured. This specification deals with the deployment and configuration of component-based applications onto distributed systems, anticipating that subcomponents might be distributed among a set of independent, interconnected nodes called domain.

In this specification, an “application” is nothing special; an application is just a component that is assumed to be independently useful. As before, this component can be implemented directly (by a monolithic implementation), or it can be implemented by an assembly, where the implementations for its subcomponents can again be either monolithic or assemblies. Ultimately, any application can be decomposed into components that have monolithic implementations. At deployment time, decisions must be made about which implementations to deploy (execute) where.

1.1 Component-Based Applications

In this specification, software components can have implementations that are either:

- compiled code (called *monolithic* implementations) or
- assemblies of other components (*assembly* implementations, providing a recursive definition)

An assembly is defined as a set of components and interconnections that *implement a component*. There is no special “top level assembly,” since assemblies are simply a method of specifying component implementations. To actually execute a component whose implementation is an assembly of lower level components, there must eventually be monolithic implementations at the “leaves” of the hierarchical implementation.

This definition of assembly means that the “application being deployed” is in fact a component. Its interface is defined as any component interface is defined. There is no special distinguished interface for “components that can be deployed as applications.” Launching a component-based application results in an object that satisfies the interface of the component interface of the “application.” Thus this specification has no need to treat the “thing being deployed” differently than a component, and enables implementation alternatives to be either monolithic compiled code artifacts or a hierarchical description of other components. This also means that any implementation, whether monolithic or assembly based, is reusable inside a larger application, *without being touched*.

A component package is a set of metadata and compiled code modules that contains implementations of a component interface. The implementations in a package can be a mix of monolithic and assembly implementations, with either or both present at any level of the hierarchy. Thus the creator of a component-based application produces a component package whose top level component interface represents the interface of the application.

Assemblies can consist of subcomponents whose implementations are inside the same package of software, or they can reference component packages that must exist in the environment outside the package containing the assembly. This not only allows packages from different vendors to be used together, but also allows dependent packages to be replaced without changing the other package or its configuration. No on-line update functionality is implied here.

To support heterogeneous systems, a package can contain more than one implementation, so that there is a choice at deployment time to find the implementation that best matches the target environment. For example, a package might contain implementations of the same component for Windows, Linux or Java.

Monolithic component implementations express requirements that must be fulfilled by properties of the system on which they will be executed, e.g. the CPU type, or available hardware. The requirements of an assembly based implementation are implied by the requirements of its subcomponents, plus additional requirements on the connections between them.

1.2 The Target Environment

The target environment is termed a domain. Domains are composed of nodes, interconnects and bridges. Nodes have computational capabilities and are a target for executing component implementations; this definition encompasses personal computers as well as SMP systems, DSPs or FPGAs. Interconnects provide a direct shared connection between nodes, e.g. representing an ethernet cable or a RapidIO fabric. Bridges route between interconnects, representing both routers and switches.

Nodes, interconnects and bridges have resources that define their features, resources and capacities. For a node, this might be the operating system type, memory or available special hardware; an interconnect might describe its bandwidth as a resource. The platform independent model does not define types of resources, it just introduces the concept. Platform specific models or domain profiles may list concrete types of resources that are relevant to the platform or the domain.

An important aspect of the target environment is that the software that supports component execution on a particular node, must be able to be implemented independently of the deployment service as a whole. This interoperability boundary allows those interested in or knowledgeable of specific types of nodes to implement deployment support for those nodes without touching the overall deployment system for the target environment.

1.3 The Deployment Process

The model in this specification is based on a process definition of deployment. The process starts after the software is developed, packaged and published by a software provider, and is acquired by the software owner, who deploys it. We call the owner at this point the *deployer.*

1.3.1 Preconditions for the Process of Deployment

Prior to deployment, the software has been packaged according to this specification, by the producer of the software, such that the metadata describing the software, and the binary compiled code artifacts, are combined into a *package*.

The package is published and somehow made available to the deployer, e.g. via a CDROM or web URL at an FTP site.

There is a *target environment*, consisting of a distributed system infrastructure (computers, networks, services), on which the software will ultimately run. There is a *repository*, which, at a minimum, is a staging area where the packaged software is captured prior to decisions about how it will run in the target environment.

1.3.2 Installation

We define *installation* as the act of taking the published software package and bringing it into a component software *repository* under the deployer's control, but the location (computer, file system, database) of this repository is not necessarily related to where the software will actually execute. It is a staging area where various policies of the deployer, such as security authentication, can be applied to the software prior to activities related to execution of the software. In the process defined here, installation is *not* related to moving software to the computers on which it will actually execute. Repositories do not necessarily need to be persistent, and they do not necessarily need to store or copy the software or metadata. Deep copy and shallow copy of the software are both supported under this specification.

1.3.3 Configuration

When the software is "in-house", in a repository, it can be functionally configured as to various default configuration options for later execution. An example would be: when this spreadsheet runs, the background color should be blue.Various configurations of a software package could be created. Configuration is *not* intended to capture the deployment decisions as to which implementation will be used or where the parts of the application will execute, but only functional configuration.

1.3.4 Planning

Planning how and where the software will run in the target environment is an activity that takes the requirements of the software to be deployed, along with the resources of the target environment on which the software will be executed, and decides which implementation and how and where the software will be run in that environment. We take care to separate this decision making step from actually acting on the decisions since there are important use cases for "advanced planning" that have *no immediate effect on the target environment*.

Advanced planning also allows for faster ultimate execution since all decisions can be made in advance (in cases where resource availability is not changing). Advanced planning can be done with an offline tool does not interact with the actual runtime environment at all, but merely “keeps score” of how it is using up the resources known to be in the target environment. Of course there are also important use cases for “just-in-time” planning, where execution follows immediately after making planning decisions based on current dynamic resource availability in the target environment.

Planning results in a *deployment plan* specific to both the software being deployed and the target environment being deployed on.

1.3.5 Preparation

Given that we define planning as deciding how and where the software will run, we define *preparation* as performing work in the target environment to be ready to execute the software, such as moving binary files to the specific computers in the target environment on which the software will execute. This work is reusable if the software is executed more than once based on the same plan. Doing this work in advance reduces the startup time when the software is actually run. Just like planning, preparation can be done “just in time”, as part of an automated scenario where the entire process happens at once.

1.3.6 Launch

Launching the application brings the application to an executing state, taking all resources that are known to be required based on the metadata in the packages. Component-based applications are launched by instantiating components, as planned, on nodes in the target environment. Launching includes interconnecting and configuring component instances, as well as starting execution. In this executing state, the application runs until it completes or is terminated via the same infrastructure that launched it.

1.3.7 All at Once, or Step by Step

This process model supports use cases where various combinations of these steps are done at different times using different tools. Of course there is the completely monolithic and automated case where a single deployment tool takes a web URL for a component package and executes it.

1.4 Relationship to the MDA

This specification is compliant with the Model Driven Architecture (MDA) defined by the OMG. It is composed of four main levels of models:

- A D&C Platform Independent Model (PIM), which constitutes the core of the specification. The D&C PIM defines the set of concepts and classes that are relevant for the implementation of the specification.

  The D&C PIM is explicitly independent of distributed component middleware technology (e.g. CORBA or J2EE), information formatting technology (e.g. XML DTD and XML), and programming languages (e.g. C++ and Java). Mappings to CORBA and XML are possible at the PSM level.

- A D&C UML profile designed to enhance the D&C PIM's readability and to facilitate the PIM-to-PSM mapping.
- A set of D&C Platform Specific Models which constitute realizations of the D&C PIM on concrete platforms. A required CCM PSM constitutes an integral part of this specification.

  A PIM-to-PSM mapping is explicitly defined for each PSM.
- A D&C Tool-Support Profile. This profile is closely related to the D&C PIM. The D&C PIM, in effect, defines the abstract syntax of a language for specifying the deployment and configuration of distributed components. The D&C Tool Support Profile defines, in effect, a concrete, UML-based syntax for this language. This concrete syntax can be employed using generic UML tools. The use of these stereotypes enables the automatic generation of D&C classes and descriptors from Deployment and Configuration UML models.

Based on the current requirements of the D&C RFP, there is no need to extend the UML metamodel at the M2 level. The use of profiles and stereotypes is sufficient to support the concepts defined in the D&C specification.

While not an explicit part of the current specification, it is also possible that different profiles of the D&C specification will be defined to satisfy the needs of different application domains, e.g. a D&C profile for web-based systems and a D&C profile for embedded systems. Because of the compatibility of the current D&C specification with the MOF 1.4, D&C profiles can be defined using the profiling mechanisms provided by UML. Such profiles would, most likely, extend the profiles defined in this specification.

Platform Independent Model 2

Contents

This chapter includes the following topics.

2.1 Segmentation of the Model

The Platform Independent Model (PIM) is segmented in two dimensions. This breaks down the overall model in a modular way such that interdependencies and complexity are minimized. The breakdown effectively creates six top level diagrams with a modest number of "external" dependencies between diagrams. The dependencies and relationships between these model segments are depicted on separate diagrams at the end of the model.

2.1.1 Dimension #1: Data Models vs. Management (or Runtime) Models.

This distinction is between a model of descriptive information, vs. the model of runtime entities that process, create, provide or store that information. In general, data models can be used to generate XML Schemas for storing and interchanging the data, and also to generate IDL data (or value) types and structures for the purpose of using the modeled data as parameters in the runtime interfaces. We use the word “management” in the sense of an active runtime entity that is dealing with (managing) the data. In general, data models are “leaves” in that they do not have intrinsic dependencies on the management/runtime models, whereas it is common for the runtime models to refer to the data models to describe parameter types in the interfaces.

In the PSMs, the IDL data structures and/or XML Schemas can be generated from the data models based on rules.

2.1.2 Dimension #2: Component Software vs. Target vs. Execution

In creating this PIM for the D&C of components, it is useful to segment the model elements according to the deployment process defined above. This should allow the different segments to be isolated according to usage (“need to know”) by actors, and then introduce (minimal) linkages or relationships between the elements as required in the different segments. This segmentation is roughly based on the process of deployment. It partitions the model with reduced/minimized interdependencies.

Component Software — output of the development, packaging, publishing processes

Component software models are about packaged component software, created by the component software development process, mostly independent of the specific target system(s) on which it will be deployed, although some requirements of the target are obviously included (compiled binary types, OS, etc.). Component software (all the packaged metadata and compiled code artifacts) is installed in a repository, configured and used for deployment planning. It exists independent of any specific target system since the planning process (and the results of the planning process) is the bridge between this information and the ultimate execution on the target.

Target Environment — where the software will run

Target models are about the computing resource environment in which a component-based application will be executed. There is static basic configuration information as well as dynamic resource (and availability) information. This is the basic “platform” on which component based applications are run, including the:

- *nodes* where software artifacts are loaded and used to instantiate components, and
- *interconnects* among nodes, to which inter-component software connections are mapped, to allow the instantiated components to intercommunicate, and
- *bridges* among interconnects. While interconnects provide a direct connection between nodes, bridges provide a routing capability between interconnects.

Interconnects are like networks or busses that multiple nodes could be attached to, and similarly, a node might be attached to multiple interconnects (like a multi-homing network host). Nodes, interconnects and bridges are collected into a *domain*, representing a particular target environment.

Execution — how the software is prepared to run, and executed based on its configuration

Execution models result from using component software models and target models to then express how component based applications will be run on a target. After creating and acquiring software, and after defining and using target information, there is planning and execution. Execution data models capture the results of planning — how the software will execute in the target environment (which implementations, running where). Execution management models use this planning information to actually prepare and launch applications. This execution happens at two levels: the whole application executing in the target environment, and the parts of the application that run on each node.

2.1.3 Summary of Model Segmentation Dimensions

Below is a table that summarizes the Data vs. Management/Runtime dimension as well as the Component Software vs. Target vs. Execution dimension. Thus the result of this segmentation can be thought of as 6 different "pages" of the model. The table below (which is not normative) summarizes the segments that are described in the next sections. PIM and PSM distinctions are weak in this summary.

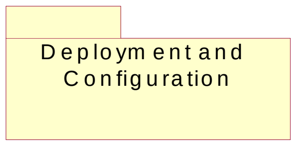
Table 2-1 D&C Model Segmentation Summary

	Data Model In PSMs, can generate XML Schemas and IDL data definitions	**Management/Runtime Model** Can imply interface IDL that may use data IDL derived from Data model. "Manager" applied to class names for consistency.	**Deployment Process Usage** How/when are the models used in the deployment process. "Tool" is used here for the client that performs and controls the process.
Component Software	**Component Data Model** of deployable component software, including descriptors for packages, interfaces, configurations, assemblies and implementations. The top-level element is the **Package-Configuration**.	**Component Management Model**: The **RepositoryManager** interface, which manages descriptive information about Component Software. Key operations include: • Install Package from URL into Repository, with name and label • Configure package, with name and label • Retrieve package configuration info by name or top level interface UUID Repository parses Component Software XML, and may be trivial in-memory (with data in IDL form only), file system based, database based. Repository can store data in persistent-IDL, XML, or private form. XML parsing can be early or late.	The software is produced and packaged according to this data model, and made available to the deployer. **Installation** tool supplies URL/location of the package to the RepositoryManager, which stores the package, *possibly* parsing, validating, authenticating etc., and creates a default configuration for the package in the repository. **Configuration** tool stores settings referring to a package, *optionally* after retrieving package information for config property validation. **Planning** tool retrieves information *in IDL data form* for decision making. Repository provides URL/location of binary artifacts so that plan need not reference repository.
Target	**Target Data Model** of the target domain, including nodes, interconnects, bridges and resources. The top level collection of this information is the **Domain**.	**Target Management Model**: The **TargetManager** interface manages **Domain** information, either offline (simply parsed from private XML) or online. It needs to allow for efficient static vs. dynamic information. Key methods: • Get base info (to allow planning tool to do preprocessing/caching of static data). • Get current info (to plan based on dynamic resource information). • Commit resources (to commit resources that are used up in the plan).	Target configuration tools can provide user interfaces to build and emit target data model XML. **Planning** tool obtains target information (in IDL data form) and creates plans. An online **TargetManager** would know and supply dynamic information collected from nodes. A **TargetManager** would initially read provided target description from XML files, and then provide the information using the data model. The TargetManager can be told about changed or new domain elements at run time.
Execution	**Execution Data Model** of decisions configuring and connecting and locating component software on a target. This is the **DeploymentPlan**.	**Target Execution Model**: The **ExecutionManager** is the runtime entity for execution of component software on the target according to the plan. Key methods: • Prepare for execution, using plan, returning **"factory"** reference (**Application Manager**) • Launch based on factory, returning **Application** reference. • Lifecycle control, using **Application** ref **NodeManager** performs the subset of execution on each node.	**Preparation** tool may parse plan XML (if not bundled with planning tool), and deliver plan in IDL-data form to Execution Manager. Thus an all-in-one tool would only have the plan in memory. **Launch** tools simply use the factory reference (Application Manager) to launch application, possibly managing the lifecycle.

The table above introduces the main elements of the platform independent model for deployment and configuration. The first column lists the three top-level data elements <ClassName>PackageConfiguration, <ClassName>Domain and <ClassName>DeploymentPlan. The second column lists the three top-level management interfaces, <ClassName>RepositoryManager, <ClassName>TargetManager and <ClassName>ExecutionManager/<ClassName>NodeManager. Each of these classes is elaborated in the upcoming sections. The third column lists use cases that are supported by this model: Installation, Configuration, Planning, Preparation and Launch. Use cases imply actors that enact them: an Administrator enacts Installation and Configuration, a Planner does the Planning, and an Executor enacts Preparation and Launch.

While the component, target and execution models are self-contained and passive, actors are the glue between them. Actors actively interface with the various management models and exchange information using the various data models. All behavior of deployment and configuration is defined by actors, as elaborated in the next chapter.

2.2 Model Diagram Conventions



This specification uses UML diagrams to show classes and their relationships. All classes are part of the Deployment and Configuration package, which contains the Components, Target, Execution, Common and Exceptions subpackages.

The Deployment and Configuration package is restricted to the MOF 1.4 subset of UML. Some non-normative diagrams from other packages are shown for explanatory purposes.

If, in a UML diagram, a class's attribute and operation compartments are suppressed, then this class is elaborated elsewhere. In this case, the diagram might also not show all of the class' associations. However, if a class is shown to have only an attribute or an operation compartment, then this signifies that the not-shown compartment is empty. I.e. if a class is shown with an attribute but no operation compartment, then the class does not have any operations.

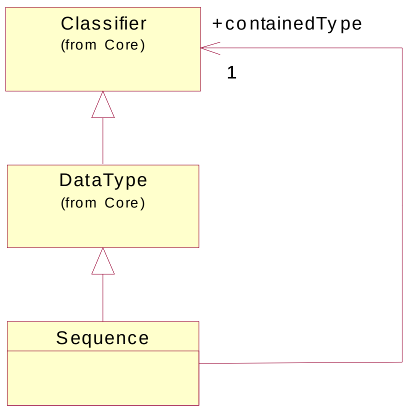
Role names on associations are made explicit wherever they are expected to appear in generated code, e.g. as an interface's attribute name. In some places, role names were added to derived associations for illustrative purposes.

Role names at the navigable end of a derived association are suppressed. Therefore, if the role name at the navigable end of an association is suppressed, the association is derived.

If a role names is suppressed at an end of an association, the name of the type at the association end, starting with a lowercase character, is used as a role name.

If an association name is suppressed, the name of the class at the source plus the name of the navigable end is used as the name of the association.

Unless otherwise mentioned, the multiplicity on the near end of navigable associations is zero to many, and the multiplicity on the near end of compositions is one to one.



This specification is aligned with MOF 1.4, which allows operation parameters to express multiplicities. However, parameter multiplicities cannot be modelled with the current version of Rational Rose. As a workaround, the Sequence metatype is introduced as above. An instance of the Sequence metatype, using the notation **Sequence(containedType)**, defines a concrete data type with no attributes but a composite navigable association to an unbounded number of elements of the contained type with the role name "**element**." For simplicity, the Sequence metaclass does not provide user-defined bounds.

Note – Instances of the Sequence metaclass are used to denote sequences of elements of the same type that are to be passed to an operation as a single parameter, or returned from an operation as a single return value, or to model multiplicities in attributes. The "`element`" role name is introduced to make the elements accessible to OCL expressions. The UML 2 Partners submission to the UML 2 RFP adds a notation to express multiplicities of parameters and return values (by putting the multiplicity in brackets). If this feature is added to UML 2.0, then this specification will be updated to the UML 2 notation, removing the Sequence kludge above.

Standard attributes are used as needed on classes for readability and identity purposes. The standard attribute names are

- `label`: A human-readable label that is not evaluated by the deployment system. It can be used to annotate classes with a user-defined string. Content is optional.
- `UUID`: A machine-readable identifier that uniquely identifies a "work product." If UUIDs compare equal, then the elements in question are considered identical. For example, if two implementation artifacts have identical UUIDs, then the deployment

system can assume that it needs to be loaded onto a node only once. UUID content is optional: a blank UUID (the empty string) is considered non-equal to any other UUID. Thus users are not required to generate proper UUIDs for internal purposes (e.g. during testing). If a UUID field is non-blank, then it must comply to the URI syntax, in alignment with MOF 2.0 identifiers.

- **name**: Names are both human-readable and machine-readable. Names are mandatory, and they must be unique within their container or context. For example, in the case of a node, the node's name must be unique within the domain.
- **location**: references an entity outside of the model. The location attribute is of type **String**, its value must comply to the URI syntax.
- **specificType**: identifies the most specific type of an interface. Components or ports with equal specificType are type equivalent. The **specificType** attribute is of type **String**; consequently, string comparison is used to compare them. PSMs define the format.
- **supportedType**: identifies all types that an interface can support. The type of this attribute is a sequence of Strings. A component or port can satisfy a requirement on any of the types listed among the supported types. The **supportedType** attribute includes the most specific type (from the **specificType** attribute) and all directly or indirectly inherited types in no particular order.

To enhance readability, in the PIM below we annotate classes with stereotypes that define two orthogonal dimensions to the class structure and relationships in the model. The first follows the Data Model vs. Management/Runtime Model dimension in the segmentation discussion above. We will use the «**Description**» and «**Manager**» stereotypes to make this distinction.

In general, «**Description**» classes generate data structures and schema, and «**Manager**» classes generate runtime interfaces.

The second annotation dimension is to identify, for «**Description**» classes, the actor in the development process for which this class a work product. These stereotypes are essentially an annotation that highlights authorship (and inherits from «**Description**», without introducing extra relationship detail in the diagrams).

Although these development actors are defined in detail later, we will briefly introduce them here:

- The «**Specifier**» specifies the interface and functional contract for components' implementations.
- The «**Implementer**» creates concrete (monolithic, coded) implementations of components including their metadata.
- The «**Packager**» creates packages (bundles) of component implementations.
- The «**Planner**» makes decisions about deployment based on target capabilities and component requirements.
- The «**DomainAdministrator**» prepares information about the target environment.

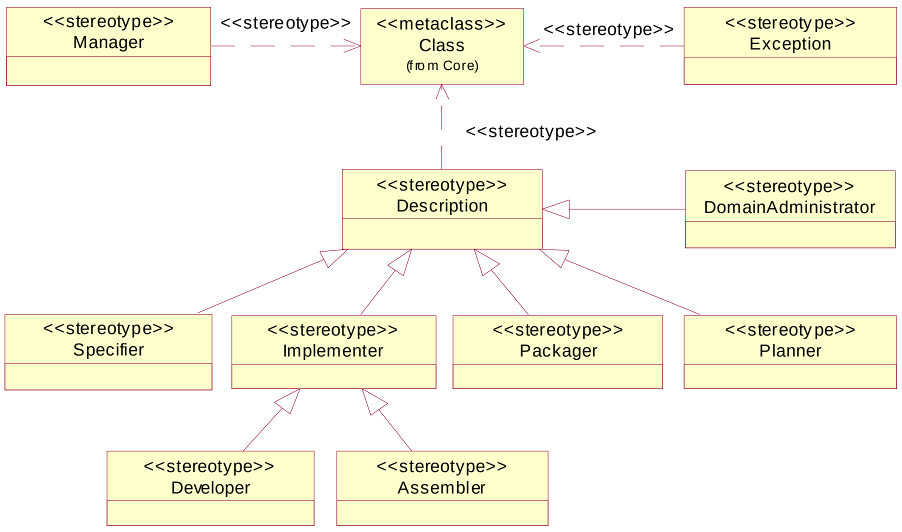
  The «**Implementer**» is in fact inherited by two derived stereotypes:

- The «**Developer**» creates monolithic (e.g., source coded/compiled) implementations.
- The «**Assembler**» creates assembly-based implementations of components.

Classes that are the work product of more than one actor are annotated with the generic «**Description**» stereotype. The creating actor can be inferred from context.

The «**Exception**» stereotype is used for exceptions that are raised by operations of management classes.

These stereotypes are represented by the "profile" diagram:



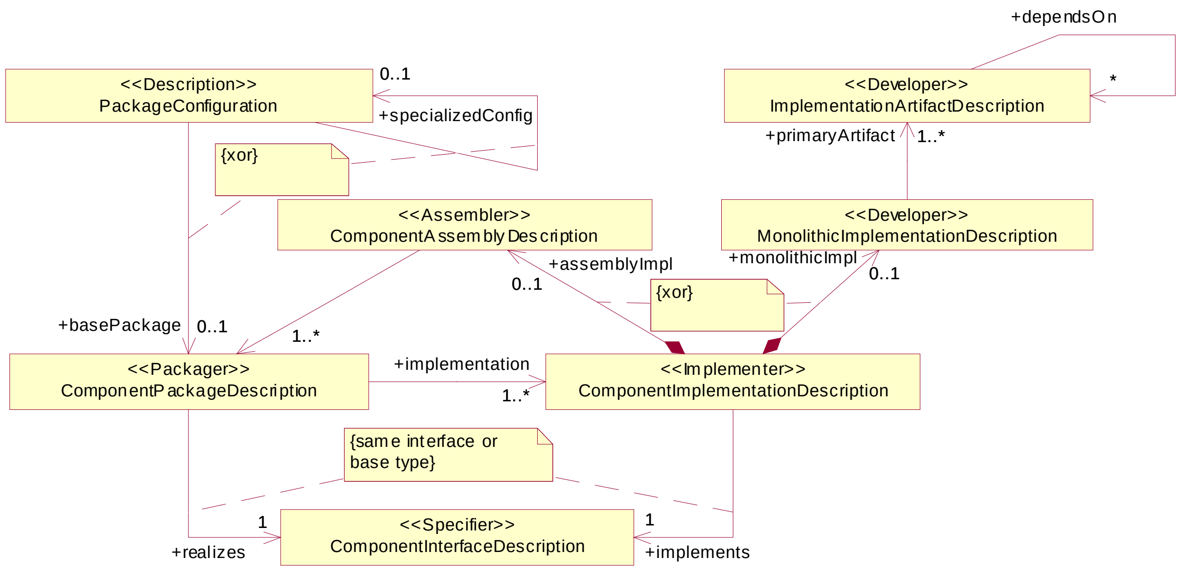
2.3 Component Data Model

The following classes are part of the Component Data Model. They are placed in the Component subpackage of the Deployment and Configuration package.

2.3.1 Component Data Model Overview

A component has an interface composed of operations, attributes and ports that may be connected to other components. A component may have a concrete (monolithic) implementation contained in an artifact (e.g., an executable file or library), or it may be recursively implemented by an assembly: a set of interconnected sub-components.

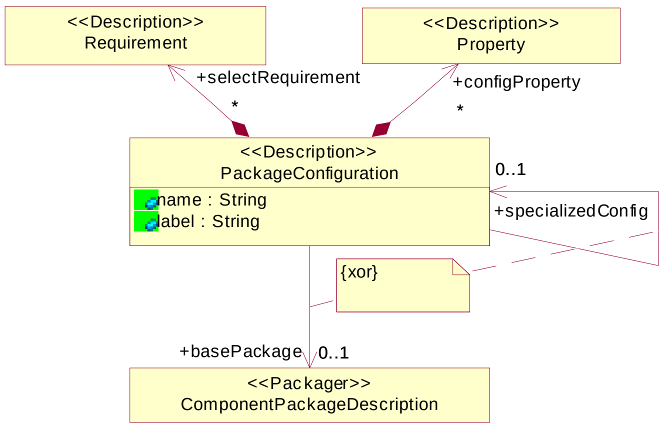
A component package contains multiple implementations of the same component. This allows distribution of a set of implementations with different properties (e.g., for different operating systems) or different hierarchies, to be distributed in a single package. Packages are installed into a repository, where they may be configured (e.g., overriding default property values) prior to deployment.



The above is an overview of the Component Data Model and represents the information about installed and configured packages provided by the <ClassName>RepositoryManager. Details about each class will be presented in the following sections.

2.3.2 PackageConfiguration

Description



A <ClassName>PackageConfiguration describes one configuration of a component package. It either specializes another <ClassName>PackageConfiguration or is directly based on a **ComponentPackageDescription**. A <ClassName>PackageConfiguration has a name, a label and two sets of properties. Configuration properties are used to configure the application's properties; their names and types must match the component's external properties. Selection requirements are used to influence deployment decisions by matching them against implementation capabilities in the **ComponentImplementationDescription**.

Attributes

name: **String**	Unique name for this <ClassName>PackageConfiguration.
label: **String**	An optional human-readable label.

Associations

specializedConfig: <ClassName>PackageConfiguration [0..1]
Links to a <ClassName>PackageConfiguration that is specialized by this <ClassName>PackageConfiguration.

basePackage: **ComponentPackageDescription** [0..1]
Links to a **ComponentPackageDescription** that this <ClassName>PackageConfiguration is based on.

selectRequirement: <ClassName>Requirement [*]
During planning, selection requirements in a <ClassName>PackageConfiguration are matched against capabilities in the **ComponentImplementationDescription**.

configProperty: <ClassName>Property [*] Properties to configure the application component with. Overrides default values in the **ComponentPackageDescription**.

Constraints

A <ClassName>PackageConfiguration must either specialize another <ClassName>PackageConfiguration or be based on a **ComponentPackageDescription**, but not both.

```
context PackageConfiguration inv:
    self.basePackage->size() = 1 xor
    self.specializedConfig->size() = 1
```

The name must be unique in the repository.

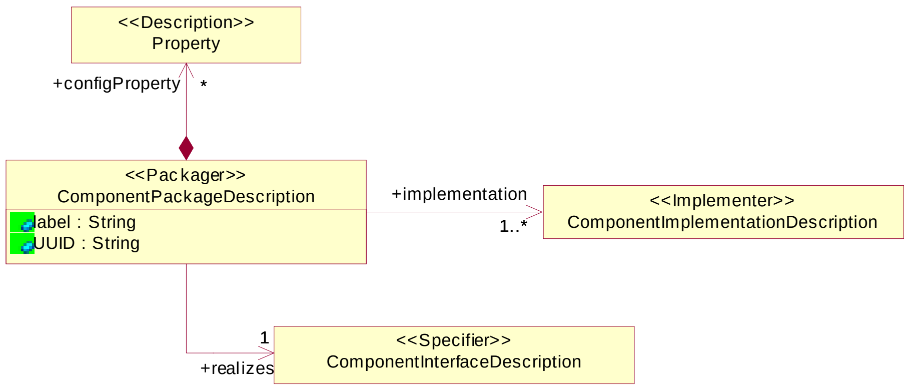
```
context PackageConfiguration inv:
    PackageConfiguration.allInstances->forAll (p1, p2 |
        p1.name = p2.name implies p1 = p2)
```

Semantics

A <ClassName>PackageConfiguration that specializes another <ClassName>PackageConfiguration extends and overrides the base configuration’s selection requirements and configuration properties. The complete set of selection requirements and configuration properties is the sum of all selection requirements and configuration properties, respectively, in the chain of <ClassName>PackageConfiguration instances, with duplicates removed.

2.3.3 ComponentPackageDescription

Description



A **ComponentPackageDescription** describes multiple alternative implementations of the same component interface. It references the interface description for the component and contains a number of configuration properties to configure the running components (which may override implementation-defined properties and which may be overridden by a <ClassName>PackageConfiguration). These configuration properties enable the packager to define default values for a component's properties regardless of which implementation for that component is chosen at deployment (planning) time.

Attributes

label: **String**	An optional human-readable label for the package.
UUID: **String**	An optional unique identifier for this package.

Associations

realizes: <ClassName>ComponentInterfaceDescription [1]
A **ComponentPackageDescription** describes implementations that realize a certain component interface.

implementation: **ComponentImplementationDescription** [1..*]
A **ComponentPackageDescription** describes multiple implementations.

configProperty: <ClassName>Property [*] These configuration properties are used to configure the component once instantiated. This allows the definition of configuration properties in a package regardless of which implementation is chosen.

Constraints

All implementations referenced by this **ComponentPackageDescription** must implement the same interface as realized by the package, or a derived interface.

```
context ComponentPackageDescription inv:
    self.implementation->forAll (
        implements.supportedType->includes (self.realizes.primaryType))
```

If the **UUID** attribute is not the empty string, then it must contain a unique identifier for the package; packages with the same non-empty **UUID** must be identical.

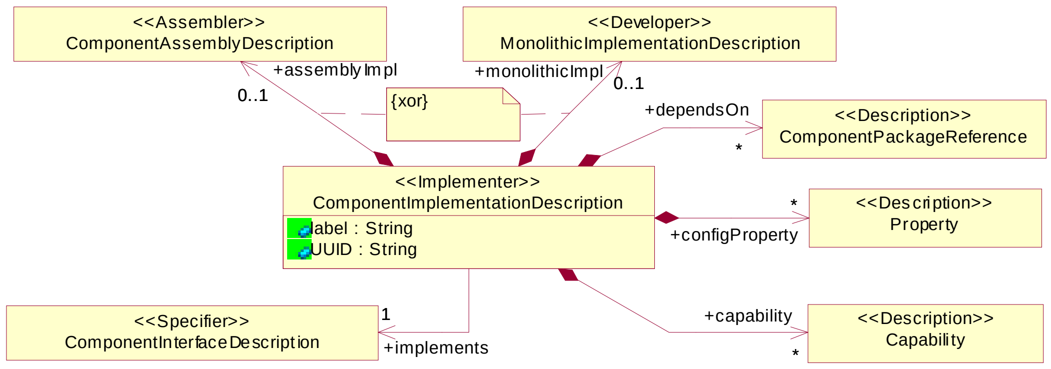
```
context ComponentPackageDescription inv:
    self.UUID <> “” implies
        ComponentPackageDescription.allInstances->forAll (p |
            p.UUID = self.UUID implies p = self)
```

Semantics

Configuration properties can be overridden in a <ClassName>PackageConfiguration. All implementations in the package are considered equally suitable for deployment, pending compatibility between implementation artifact requirements and node resources, and selection properties required by a <ClassName>PackageConfiguration.

2.3.4 ComponentImplementationDescription

Description



<<Assembler>> ComponentAssemblyDescription
<<Developer>> MonolithicImplementationDescription
+assemblyImpl
0..1
+monolithicImpl
0..1
{xor}
+dependsOn
*
<<Description>> ComponentPackageReference
<<Implementer>> ComponentImplementationDescription
label : String
UUID : String
*
+configProperty
<<Description>> Property
<<Specifier>> ComponentInterfaceDescription
1
+implements
+capability
*
<<Description>> Capability

A **ComponentImplementationDescription** describes a specific implementation of a component interface. This implementation can be either assembly based or monolithic. The **ComponentImplementationDescription** may contain configuration properties that are used to configure each component instance ("default values"). Implementations may be tagged with user-defined capabilities. Administrators can then select among implementations using selection requirements in a <ClassName>PackageConfiguration; Assemblers can place requirements on implementations in a **SubcomponentInstantiationDescription**.

Attributes

label: **String**	An optional human-readable label for the implementation.
UUID: **String**	An optional unique identifier for this implementation.

Associations

implements: <ClassName>ComponentInterfaceDescription [1]
The component interface implemented by this implementation.

assemblyImpl: **ComponentAssemblyDescription** [0..1]
In case of an assembly based implementation, this describes the assembly.

monolithicImpl: <ClassName>MonolithicImplementationDescription [0..1]
In case of a monolithic implementation, this describes the monolithic implementation.

configProperty: <ClassName>Property [*] These are implementation specific configuration properties that are used to configure the component once instantiated.

capability: <ClassName>Capability [*] These are tags that a <ClassName>PackageConfiguration can match against to discriminate between implementations.

dependsOn: **ComponentPackageReference** [*]
Expresses a dependency on other packages; Implementations of referenced packages must be deployed in the target environment before this package can be deployed.

Constraints

An implementation is either assembly based or monolithic, consequently there must be either a **ComponentAssemblyDescription** or a <ClassName>MonolithicImplementationDescription, but not both.

```
context ComponentImplementationDescription inv:
    self.assemblyImpl.size() = 1 xor
    self.monolithicImpl.size() = 1
```

If the **UUID** attribute is not the empty string, then it must contain a unique identifier for the implementation; implementations with the same non-empty **UUID** must be identical.

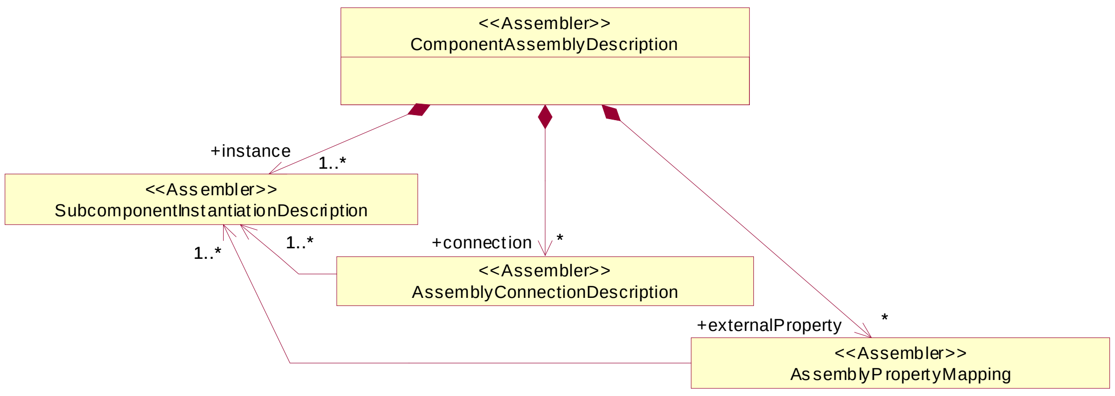
```
context ComponentImplementationDescription inv:
    self.UUID <> “” implies
        ComponentImplementationDescription.allInstances->forAll (i |
            i.UUID = self.UUID implies i = self)
```

Semantics

Configuration properties can be overridden in a **ComponentPackageDescription** or in a <ClassName>PackageConfiguration.

2.3.5 ComponentAssemblyDescription

Description



In the case of an assembly based implementation, the **ComponentAssemblyDescription** contains information about sub-component instances (**SubcomponentInstantiationDescription**), connections among ports (<ClassName>AssemblyConnectionDescription), and about the mapping of the assembly's properties (i.e. of the component that the assembly is implementing) to properties of its subcomponents.

Attributes

No attributes.

Associations

instance: **SubcomponentInstantiationDescription** [1..*]
Describes instances of subcomponents.
connection: <ClassName>AssemblyConnectionDescription [*]
Describes connections between ports.
externalProperty: <ClassName>AssemblyPropertyMapping [*]
Maps the external properties of the component that is implemented by the assembly to properties of subcomponent instances.

Constraints

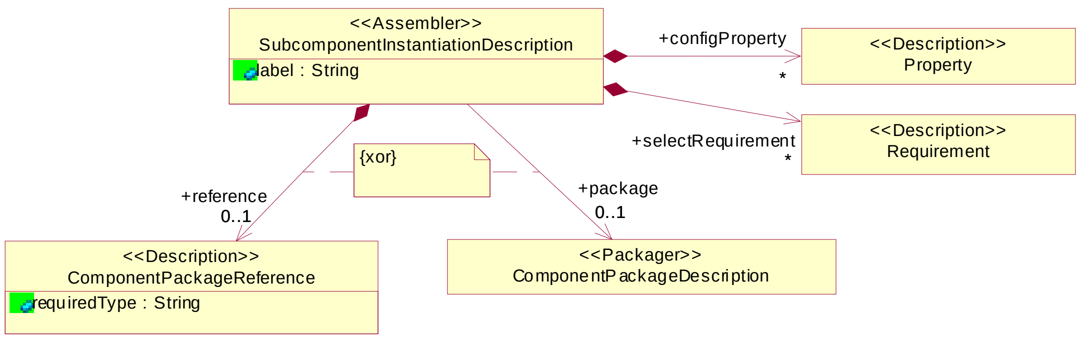
No constraints.

Semantics

An assembly is composed of components and itself implements a component, as implied by the **ComponentImplementationDescription** that this **ComponentAssemblyDescription** is contained in. The component being implemented by the assembly is referred to as the "external component" of the assembly. Connections exist among the subcomponents' ports and the external component's ports, similar to a wiring diagram in circuit design, where a circuit is designed by wiring chips among themselves and wiring them to external pins.

2.3.6 SubcomponentInstantiationDescription

Description



In an assembly based implementation, the **SubcomponentInstantiationDescription** describes one instance of a sub-component.

The **SubcomponentInstantiationDescription** links to a package that provides implementations for the sub-component that is to be instantiated. There is either a link to a **ComponentPackageDescription** in case a package recursively contains packages for its sub-components, or there is a link to a **ComponentPackageReference** that contains the **requiredType** of a component interface. Users of the Component Data Model will have to contact a repository (possibly via a search path) in order to find a package that implements this interface.

Attributes

label: **String**	An optional human-readable label for the subcomponent.

Associations

package: **ComponentPackageDescription** [0..1]	Describes a package that provides an implementation for this sub-component instance.
reference: **ComponentPackageReference** [0..1]	References an outside package that provides an implementation for this subcomponent instance.
configProperty: <ClassName>Property [*]	Configuration properties that are used to configure the subcomponent instance when the assembly is instantiated.

Constraints

There can be either a package or a reference, but not both.

```
context SubcomponentInstantiationDescription inv:
    self.reference->size() = 1 xor
    self.package->size() = 1
```

Semantics

The planner will consider the implementations in the package that is either contained or referenced and select the implementation that is used to instantiate the subcomponent based on compatibility and preferences. Configuration properties for subcomponents are final, they can only be overridden if mapped to an external port of the component that this assembly is implementing. A **SubcomponentInstantiationDescription** does not have any deployment requirements of its own, since a specific implementation for the subcomponent will be selected by the planner.

2.3.7 ComponentPackageReference

Description

References an outside package that provides an implementation for this subcomponent instance.

Attributes

requiredType: **String**	Identifies a required component interface by type. The implementation that is chosen to satisfy the reference must support this type.

Associations

No associations.

Constraints

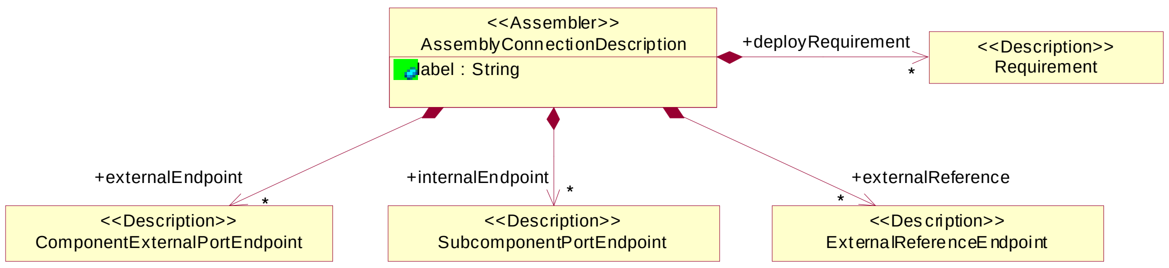
No constraints.

Semantics

The planner will use the **requiredType** to search repositories for appropriate package configurations and then select an implementation from that package to instantiate a subcomponent from.

2.3.8 AssemblyConnectionDescription

Description



An <ClassName>AssemblyConnectionDescription element describes a connection that is to be made among ports within an assembly. A connection can be thought of as a single path in a circuit wiring diagram with multiple endpoints. In this analogy, a signal that is sent onto the path is received by all receiving endpoints. There are three different types of endpoints, the most obvious being the <ClassName>SubcomponentPortEndpoint, which reflects a connection to the port of a subcomponent within the assembly. The <ClassName>ComponentExternalPortEndpoint reflects a connection to an external port of the component that is implemented by the assembly. The <ClassName>ExternalReferenceEndpoint reflects a connection to a location outside the assembly by URL (e.g., using a corbaname reference).

Some deployment requirements may be associated with the connection information; these requirements must be satisfied by the interconnect(s) in the target model over which the connection is routed at deployment time. PSMs and domain specific profiles will define a vocabulary for deployment requirements.

A label can optionally be associated with the <ClassName>AssemblyConnectionDescription. Assembly design tools might use this label to visualize the connection.

Attributes

label: **String**	An optional human-readable identifier for this connection. May be used by visual design tools.

Associations

deployRequirement: <ClassName>Requirement [*]
These connection requirements must be satisfied by the interconnects over which the connection is routed.

internalEndpoint: <ClassName>SubcomponentPortEndpoint [*]
Identifies a port of a component within the assembly as an endpoint of this connection.

externalEndpoint: <ClassName>ComponentExternalPortEndpoint [*]
Identifies a port of the component that is implemented by the assembly as an endpoint of this connection.

externalReference: <ClassName>ExternalReferenceEndpoint [*]
Identifies a location outside the assembly as an endpoint of this connection.

Constraints

The number of endpoints to a connection must be at least two.

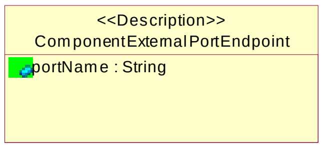
```
context AssemblyConnectionDescription inv:
    Set{self.externalEndpoint,
        self.internalEndpoint,
        self.externalReference}->size() >= 2
```

Semantics

At assembly design time, the compatibility of the endpoints can be verified based on the information known about the endpoints, e.g., appropriate user, provider, multiplex semantics. At planning time, compatibility of the connection's requirements with the resources of the interconnects that the connection is routed over will be verified. At execution time, connections between the endpoints will be established.

2.3.9 ComponentExternalPortEndpoint

Description



Identifies a port of the external component as an endpoint of the connection described by the <ClassName>AssemblyConnectionDescription that this element is contained in.

Attributes

portName: **String** The name of the port of the external component.

Associations

No associations.

Constraints

The port name must be valid for the external component.

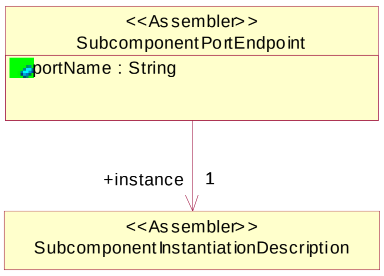
```
context ComponentExternalPortEndpoint inv:
    let if = self.assemblyConnectionDescription.
                componentAssemblyDescription.
                componentImplementationDescription.
                implements
    if.port->exists (p | p.name = self.portName)
```

Semantics

See above.

2.3.10 SubcomponentPortEndpoint

Description



Identifies a port of a component within the assembly as an endpoint of the connection described by the <ClassName>AssemblyConnectionDescription that this element is contained in.

Attributes

portName: **String** The name of the port of the associated subcomponent instance that is to be an endpoint of this connection.

Associations

instance: **SubcomponentInstantiationDescription** [1]
The associated subcomponent instance.

Constraints

The port name must be valid for the referenced component.

```
context SubcomponentPortEndpoint inv:
    self.instance.package->size() = 1 implies
      self.instance.package.interface.port.exists (name = self.portName)
```

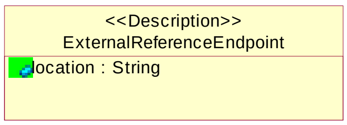
If the **SubcomponentInstantiationDescription** references a package instead of containing it (i.e., if it contains a **ComponentPackageReference**), then the constraint cannot be expressed within the repository but must be checked by the Planner.

Semantics

See above.

2.3.11 ExternalReferenceEndpoint

Description



Identifies a location outside the assembly as an endpoint of the connection described by an <ClassName>AssemblyConnectionDescription.

Attributes

location: **String**	References a port outside of the assembly that is to be an endpoint of this connection, which is resolved at execution time.

Associations

No associations.

Constraints

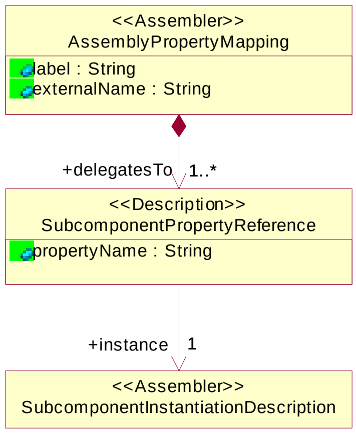
No constraints.

Semantics

The location is to be an endpoint to this connection in the assembly. Whether the endpoint is a provider or user port is implied by the URL, and its type is assumed to be compatible with the connection.

2.3.12 AssemblyPropertyMapping

Description



<ClassName>AssemblyPropertyMapping is part of the **ComponentAssemblyDescription**. It identifies a property of the external component and the subcomponents' properties that it delegates to.

Attributes

label: **String**	An optional human-readable label for this mapping.
externalName: **String**	The name of a property of the external component.

Associations

delegatesTo: <ClassName>SubcomponentPropertyReference [1..*]

References ports of subcomponents within the assembly that the property is delegated (or propagated) to.

Constraints

The **externalName** must match the name of a property of the external component.

Semantics

If the component's property is configured, the configuration value will be delegated (propagated) to the specified subcomponent ports in the assembly.

2.3.13 SubcomponentPropertyReference

Description

Identifies a property of a component within the assembly or deployment plan that an property of the external component delegates to.

Attributes

propertyName: **String**	The name of the property of that subcomponent instance that the external property is delegated to.

Associations

instance: **SubcomponentInstantiationDescription** [1]

The associated subcomponent instance.

Constraints

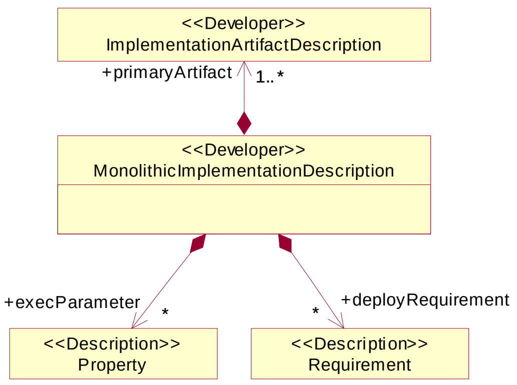
The **propertyName** must match the name of a property of the referenced subcomponent.

Semantics

No semantics.

2.3.14 MonolithicImplementationDescription

Description



In the case of a monolithic implementation, the <ClassName>MonolithicImplementationDescription describes the artifacts that are involved in this implementation. It references primary implementation artifacts (that may then depend on other supporting implementation artifacts). There may be some requirements associated with the monolithic implementation that are matched against node resources during deployment. The author of the implementation may associate some execution parameter properties with the implementation as hints to the target environment about the instantiation of the component (e.g., search path settings, environment variables). Some execution parameters may also relate to primary artifacts (e.g., entry points).

Attributes

No attributes.

Associations

execParameter: <ClassName>Property [*] Execution parameters that are passed to the target environment.

deployRequirement: <ClassName>Requirement [*]
Requirements that are matched against node resources during planning.

primaryArtifact: <ClassName>ImplementationArtifactDescription [1..*]
The primary implementation artifacts.

Constraints

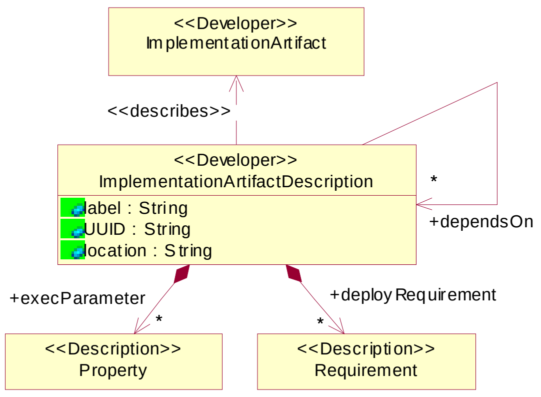
No constraints.

Semantics

Execution parameters are evaluated by the target environment and may include hints about how to instantiate a component from the implementation artifacts.

2.3.15 ImplementationArtifactDescription

Description



The <ClassName>ImplementationArtifactDescription describes an implementation artifact that is associated with a monolithic component implementation. It contains an reference to the location of the implementation artifact and may refer to other <ClassName>ImplementationArtifactDescription elements that this implementation artifact depends on (e.g., shared libraries or support files). The <ClassName>ImplementationArtifactDescription may contain deployment requirements that must be matched by a node's resources during deployment. The <ClassName>ImplementationArtifactDescription also contains execution parameters that are relevant to the target node's infrastructure (e.g., command line parameters).

Attributes

label: String	An optional human-readable label.
UUID: String	An optional unique identifier for this artifact.
location: String	The location of the implementation artifact.

Associations

dependsOn: <ClassName>ImplementationArtifactDescription [*]
References other <ClassName>ImplementationArtifactDescription elements for implementation artifacts that this implementation artifact depends on.

execParameter: <ClassName>Property [*] Execution parameters with hints to the target environment about the execution of this implementation artifact.

deployRequirement: <ClassName>Requirement [*]
Requirements that are matched against node resources.

Constraints

If the **UUID** field is non-empty, then it must contain a unique identifier for the artifact; artifacts with the same non-empty **UUID** must be identical.

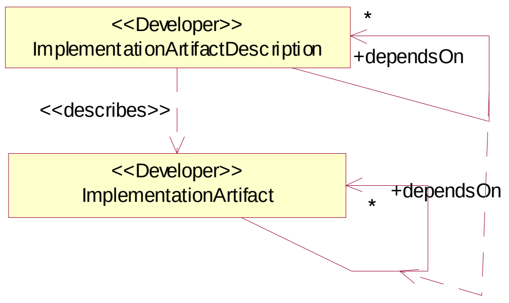
```
context ImplementationArtifactDescription inv:
    self.UUID <> “” implies
        ImplementationArtifactDescription.allInstances->forAll (i |
            i.UUID = self.UUID implies i = self)
```

Semantics

All dependent implementation artifacts have to be installed on (or available to) a node before a component can be instantiated from them.

2.3.16 ImplementationArtifact

Description



An <ClassName>ImplementationArtifact is a (potentially complete) piece of a concrete component implementation. An <ClassName>ImplementationArtifact is opaque to the deployment process and can only be evaluated in the context of a target environment (e.g., for execution). The <ClassName>ImplementationArtifactDescription captures the properties of an <ClassName>ImplementationArtifact that are relevant to the deployment process.

Attributes

No attributes.

Associations

No associations.

Constraints

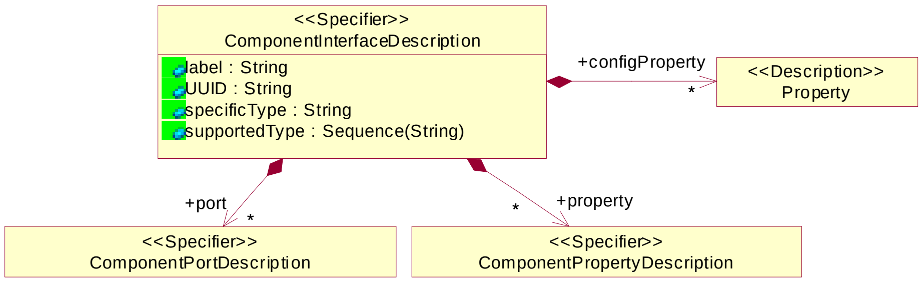
No constraints.

Semantics

The dependency relationship between <ClassName>ImplementationArtifactDescription elements reflects the dependency between implementation artifacts (e.g., executables depending on shared libraries) in the data model.

2.3.17 ComponentInterfaceDescription

Description



<ClassName>ComponentInterfaceDescription describes a component's interface. This information can be used by e.g. an assembly tool to verify interface compatibility. The component interface is identified by a unique identifier. A component has properties and ports.

Attributes

label: String	An optional human-readable label for this interface.
UUID: String	An optional unique identifier for this interface.
specificType: String	The most specific type supported by this component interface.
supportedType: Sequence(String)	Component interface types supported by this interface (e.g., by inheritance).

Associations

port: <ClassName>ComponentPortDescription [*]	Describes the ports of this component interface.
property: <ClassName>ComponentPropertyDescription [*]	Identifies the configurable properties of a component interface.
configProperty: <ClassName>Property [*]	Optional default values for properties.

Constraints

The supported types must include the specific type.

```
context ComponentInterfaceDescription inv:
    self.supportedType->includes (self.specificType)
```

If the **UUID** field is non-empty, then it must contain a unique identifier for the interface; interfaces with the same non-empty **UUID** must be identical.

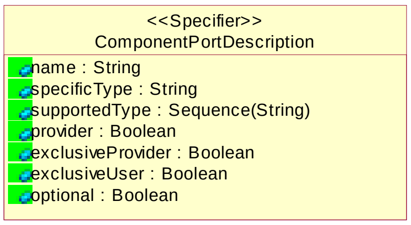
```
context ComponentInterfaceDescription inv:
    self.UUID <> "" implies
        ComponentInterfaceDescription.allInstances->forAll (i |
            i.UUID = self.UUID implies i = self)
```

Semantics

Default configuration values can be overridden by assemblies, implementations, packages or package configurations.

2.3.18 ComponentPortDescription

Description



<ClassName>ComponentPortDescription describes a port within a component interface. Tools can use this information to e.g., verify port compatibility in connections.

Attributes

name: **String**	The name of the port.
specificType: **String**	The most specific type supported by the port.
supportedType: **sequence(String)**	All types supported by this port, including the specific and inherited

	types. All of the types listed in this attribute are acceptable for a connection.
provider: **Boolean**	Identifies whether the port acts in the role of provider or user, for any connection attached to it.
exclusiveProvider: **Boolean**	If set to true, then this port expects that there is at most one provider on the connection that it is an endpoint to.
exclusiveUser: **Boolean**	If set to true, then this port expects that there is at most one user on the connection that it is an endpoint to.
optional: **Boolean**	Identifies whether connecting this port is optional or mandatory.

Associations

No associations.

Constraints

The supported types must include the specific type.

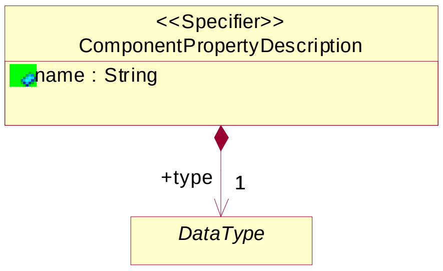
```
context ComponentPortDescription inv:
    self.supportedType->includes (self.specificType)
```

Semantics

Ports that are endpoints of a connection must support the same type (protocol). Endpoints to a connection can act in the role of either provide or user. For user or provider ports, if **exclusiveProvider** is true, then the connection may not have more than one provider port as an endpoint; if **exclusiveUser** is true, then at most one user port may be an endpoint. For both provider and user ports, if optional is true, then it is not mandatory to use this port as an endpoint to any connection. Thus any implementations would have to function when there was no connection.

2.3.19 ComponentPropertyDescription

Description



<ClassName>ComponentPropertyDescription describes a component property.

Attributes

name: **String**	The name of the property.

Associations

type: <ClassName>DataType [1] The data type of this property.

Constraints

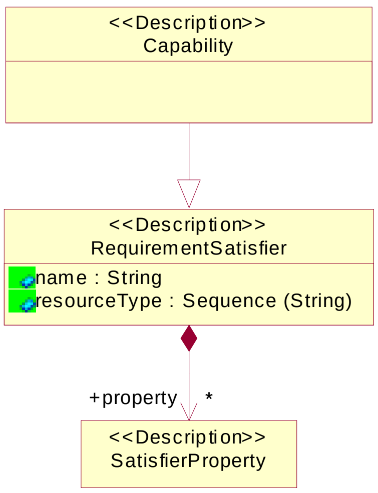
No constraints.

Semantics

If this property is configured, the value must conform to the type.

2.3.20 Capability

Description



<ClassName>Capability is used within the **ComponentImplementationDescription** to describe an implementation’s capabilities, which are matched against selection requirements in **SubcomponentInstantiationDescription** or <ClassName>PackageConfiguration. It extends the <ClassName>RequirementSatisfier class, but does not add any attributes or associations.

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

Capabilities are not consumable. <ClassName>SatisfierProperty elements that are part of <ClassName>Capability cannot use the “**Quantity**” or “**Capacity**”<ClassName>SatisfierPropertyKind kinds.

```
context Capacity inv:
    self.property->forAll (
        kind <> SatisfierPropertyKind::Quantity and
        kind <> SatisfierPropertyKind::Capacity)
```

Semantics

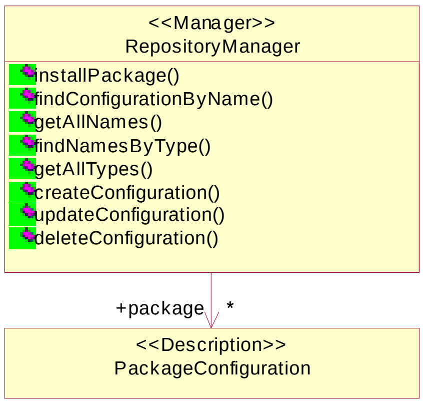
Same as for <ClassName>RequirementSatisfier.

2.4 Component Management Model

The <ClassName>RepositoryManager class is placed in the Component subpackage of the Deployment and Configuration package.

2.4.1 RepositoryManager

Description



A <ClassName>RepositoryManager manages component data. It maintains a collection of <ClassName>PackageConfiguration elements. Package installation results in a new **ComponentPackageDescription** represented by a <ClassName>PackageConfiguration with an empty set of properties. <ClassName>PackageConfigurations are identified by labels that are unique within the repository. The <ClassName>RepositoryManager can provide a list of all package configuration labels that support a given component interface's UID, and a list of all UIDs. <ClassName>PackageConfiguration elements can be created based on (specializing) existing package configurations or by installing a new package. After creation, package configurations can be updated.

Operations

installPackage (**name**: **String**, **label**: **String**, **location**: **String**)

Installs a package in the repository, assigning the given name and label to the new <ClassName>PackageConfiguration. Raises the <ClassName>NameExists exception if a configuration by this name already exists. Raises the <ClassName>PackageError exception if an internal error is detected in the package.

findConfigurationByName (**name**: **String**): <ClassName>PackageConfiguration

Locates a <ClassName>PackageConfiguration by name. Raises the <ClassName>NoSuchName exception if the name does not exist.

getAllNames (): **Sequence(String)**

Returns a list of all package configuration names.

findNamesByType (**type**: **String**): **Sequence(String)**

Finds all configurations of packages that support the given interface type. Returns a sequence of names.

getAllTypes (): **Sequence(String)**

Returns a sequence of all interface types for which packages are available.

createConfiguration (**nname**: **String**, **bname**: **String**, **cp**: **Sequence(**<ClassName>Property**),**
sr: **Sequence(<ClassName>Requirement)**)

Creates a new <ClassName>PackageConfiguration based on an existing configuration, extending and overriding the base's selection requirements and configuration properties. Raises the <ClassName>NoSuchName exception if the base name does not exist. Raises the <ClassName>NameExists exception if the new name already exists.

updateConfiguration (**name**: **String**, **cp**: **Sequence(**<ClassName>Property**),**
sr: **Sequence(<ClassName>Requirement)**)

Updates an existing <ClassName>PackageConfiguration, extending and replacing its selection requirements and configuration properties. Raises the <ClassName>NoSuchName exception if the name does not exist.

deleteConfiguration (**name**: **String**, **deletePackage**: **Boolean**)

Deletes the <ClassName>PackageConfiguration that is referenced by name and all other <ClassName>PackageConfiguration elements that are based on it. If this is the last <ClassName>PackageConfiguration for a component package, and if the **deletePackage** parameter is set to true, then the package is also removed from the repsoitory. Raises the <ClassName>LastConfiguration exception if this is the last <ClassName>PackageConfiguration for a component package and the **deletePackage** parameter is false. Raises the <ClassName>NoSuchName exception if the name does not exist.

Associations

package: <ClassName>PackageConfiguration [*]

A <ClassName>RepositoryManager manages a number of package configurations.

Constraints

No constraints.

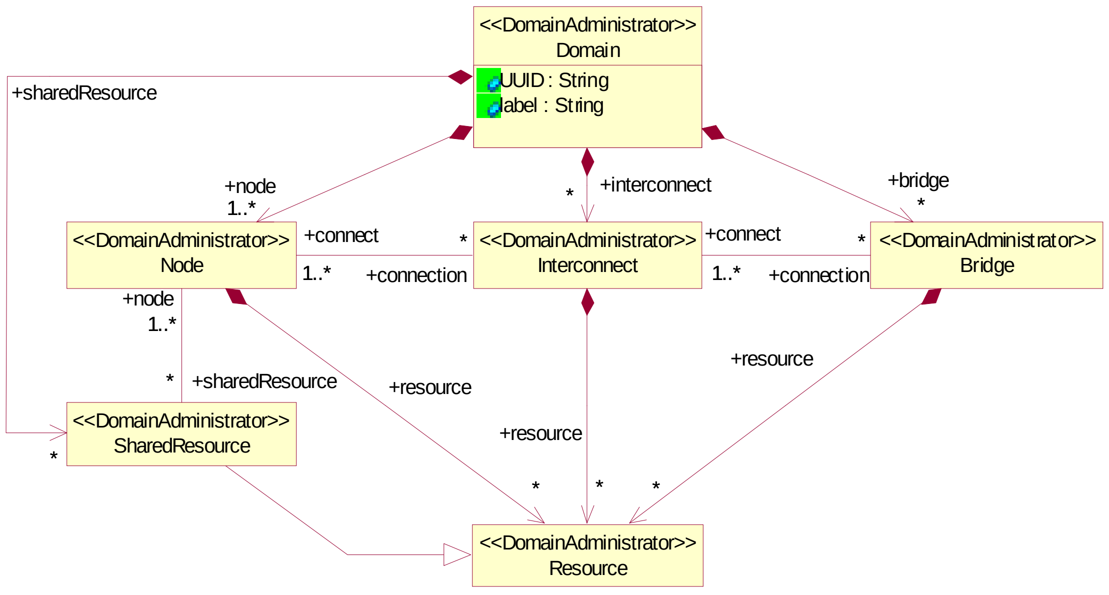
Semantics

No additional semantics.

2.5 Target Data Model

The following classes are part of the Target Data Model. They are placed in the Target subpackage of the Deployment and Configuration package.

The Target Model describes and manages information about the domain into which applications can be deployed. A domain is a set of interconnected nodes with bridges routing between interconnects. Shared resources are logically contained in the domain itself.



The top-level entity of target information is the <ClassName>Domain. A <ClassName>Domain is composed of <ClassName>Node, <ClassName>Interconnect, <ClassName>Bridge and <ClassName>SharedResource elements. Nodes have computational capabilities and are targets for the execution of component instances. Nodes may have resources and be associated with shared resources. While resources belong to the node, a shared resource may be shared between nodes. Artifact requirements must be satisfied by the resources and shared resources of the node that it is to be installed on.

Interconnects provide direct connections among nodes. They have resources but no shared resources. Interconnects are targets for the deployment of connections between components. Connection requirements must be satisfied by the interconnect's resources. Bridges route between interconnects and therefore provide indirect connections between nodes. Connections use some combination of the resources of interconnects and bridges to accomplish the communication between connected ports of instances.

The above is an overview of the Target Data Model. Details about each class in the Target Data Model will be presented in the following sections.

2.5.1 Domain

Description

The <ClassName>Domain is the container that wraps information about its <ClassName>Node, <ClassName>Interconnect, <ClassName>Bridge, and <ClassName>SharedResource elements. It represents the entire target environment.

Attributes

label: **String**	An optional human-readable label for the domain.
UUID: **String**	An optional unique identifier for this domain.

Associations

node: <ClassName>Node [1..*]	<ClassName>Node elements that belong to the domain.
interconnect: <ClassName>Interconnect [*]	<ClassName>Interconnect elements that provide direct connections between nodes.
bridge: <ClassName>Bridge [*]	<ClassName>Bridge elements route between interconnects and therefore provide indirect connections between nodes.
sharedResource: <ClassName>SharedResource [*]	Shared resources that belong to the domain.

Constraints

The top-level elements in a domain all have `name` attributes. These names must be unique within the domain.

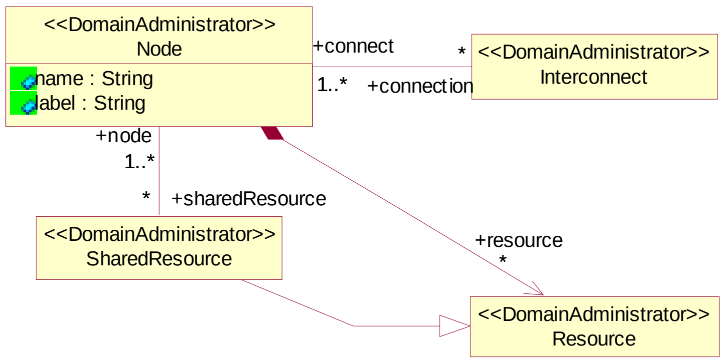
```
context Domain inv:
    let elements = Set {self.node, self.interconnect,
                        self.bridge, self.sharedResource}
    elements->forAll (e1, e2 | e1.name = e2.name implies e1 = e2)
```

Semantics

No additional semantics.

2.5.2 Node

Description



Nodes are connected to zero or more interconnects that enable components that are instantiated on this node to communicate with components on other nodes. Nodes may own resources and may have access to shared resources that are shared between nodes.

Attributes

name: **String**	The node's name.
label: **String**	An optional human readable label for the node.

Associations

connection: <ClassName>Interconnect [*] A node may be connected to interconnects.
resource: <ClassName>Resource [*] A node may have resources.
sharedResource: <ClassName>SharedResource [*]
A node may have access to shared resources.

Constraints

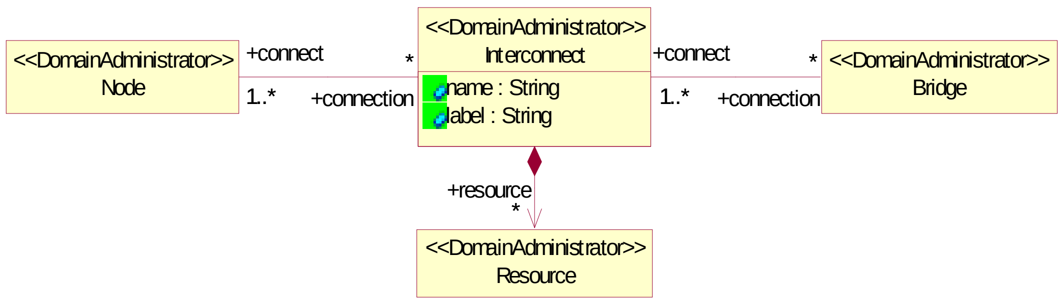
The `name` of the <ClassName>Node must be unique within the <ClassName>Domain (see above).

Semantics

A node's resources and shared resources are matched against implementation requirements.

2.5.3 Interconnect

Description



An <ClassName>Interconnect provides a shared direct connection between one or more nodes. It has resources, but no shared resources. Resources are matched against a connection's requirements (from the <ClassName>AssemblyConnectionDescription) at deployment time.

An <ClassName>Interconnect that is attached to only a single node can be used to describe the loopback connection. A loopback connection is implicit; components can always be interconnected locally. Sometimes, it may be useful or necessary to describe the type(s) of available loopback connections (e.g., “shared memory”), or their resources or capabilities (e.g., latency).

Attributes

name: **String**	The interconnect’s name.
label: **String**	An optional human-readable label for the interconnect.

Associations

connect: <ClassName>Node [1..*]	The nodes that this interconnect provides a connection in be-tween.
connection: <ClassName>Bridge [*]	The bridges that provide connectivity to other interconnects.
resource: <ClassName>Resource [*]	Interconnects have resources.

Constraints

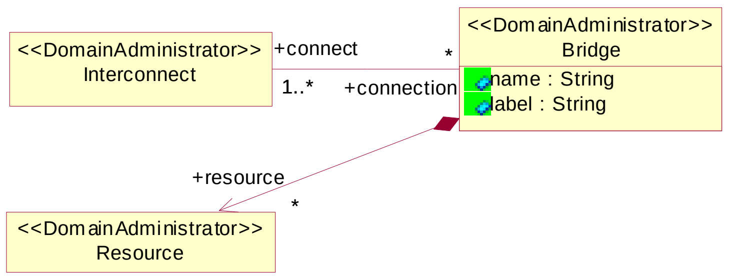
The **name** must be unique within the domain (see above).

Semantics

An interconnect’s resources are matched against connection requirements.

2.5.4 Bridge

Description



A <ClassName>Bridge exists between interconnects to describe an indirect communication path between nodes. If a connection is to be deployed between components that are instantiated on nodes that are not directly connected, therefore requiring bridging, the connection's requirements must be satisfied by the resources of each interconnect and bridge in between.

Attributes

name: **String**	The bridge's name.
label: **String**	An optional human-readable label for this bridge.

Associations

connect: <ClassName>Interconnect [1..*]	The interconnects that this bridge provides connectivity between.
resource: <ClassName>Resource [*]	Bridges have resources.

Constraints

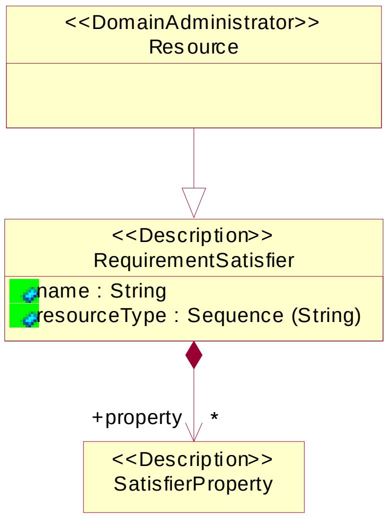
The `name` must be unique within the domain (see above).

Semantics

A bridge's resources are matched against connection requirements.

2.5.5 Resource

Description



<ClassName>Resource elements express <ClassName>Node, <ClassName>Interconnect and <ClassName>Bridge features within the target environment. They are matched against implementation requirements at planning time. <ClassName>Resource extends the <ClassName>RequirementSatisfier class, but does not add any attributes or associations.

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

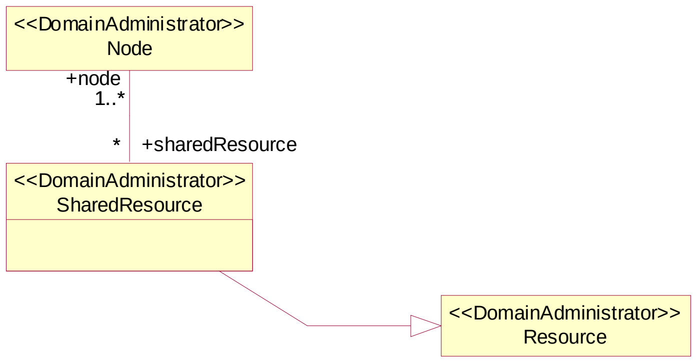
The name of a resource must be unique within the container.

Semantics

Same as for <ClassName>RequirementSatisfier.

2.5.6 SharedResource

Description



Shared resources are resources that are shared between nodes. They are semantically equivalent to "normal" resources; however, the planner must make sure that a shared resource is not exhausted by using it from multiple nodes in parallel.

Attributes

No additional attributes.

Associations

nodes: <ClassName>Node [1..*]	The nodes that have access to this <ClassName>SharedResource.

Constraints

The **name** of the <ClassName>SharedResource must be unique within the domain (see above).

Semantics

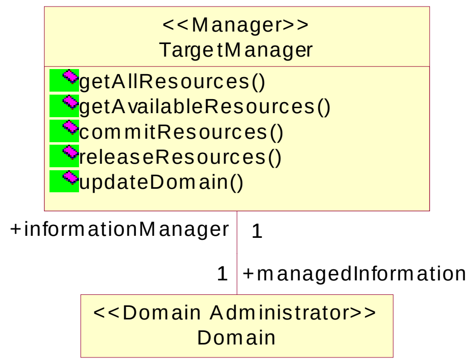
Same as for <ClassName>Resource and for <ClassName>RequirementSatisfier.

2.6 Target Management Model

The <ClassName>TargetManager and <ClassName>DomainUpdateKind classes are placed in the Target subpackage of the Deployment and Configuration package.

2.6.1 TargetManager

Description



The <ClassName>TargetManager provides information about the <ClassName>Domain using the Target Data Model and tracks resource usage within the domain. Note that this specification limits the features of the <ClassName>TargetManager to those related to deployment. While domains and nodes may have properties, exposing an interface to configure them is out of the scope of this specification.

Operations

getAllResources (): <ClassName>Domain
Returns static information about the domain, with resources at their full capacity.

getAvailableResources (): <ClassName>Domain
Returns online information about the domain; resources will reflect their remaining capacity.

commitResources (**plan**: <ClassName>DeploymentPlan)
Commits resources that are used by the instantiation of an application from a deployment plan. Raises the <ClassName>ResourceNotAvailable exception if one of the requirements cannot be satisfied. Raises the <ClassName>PlanError exception if the plan cannot be processed due to an inconsistency.

releaseResources (**plan**: <ClassName>DeploymentPlan)
Releases resources that are used by the instantiation of an application from a deployment plan. Raises the <ClassName>PlanError exception if the plan cannot be processed due to an inconsistency.

updateDomain (**elements**: **Sequence(String)**, **domainSubset**: <ClassName>Domain, `updateKind`: <ClassName>DomainUpdateKind)
Updates <ClassName>Domain information within the <ClassName>TargetManager. The elements parameter identifies the names of nodes, interconnects, bridges and shared resources to be updated. The `domainSubset` contains information about the ele-

ments and their associations. The `updateKind` identifies whether the elements are to be added, deleted or updated.

Associations

managedInformation: <ClassName>Domain [1]

A <ClassName>TargetManager manages information about a single <ClassName>Domain.

Constraints

No constraints

Semantics

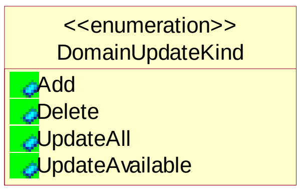
Resources are centrally managed by the <ClassName>TargetManager, it is assumed that the <ClassName>TargetManager has complete knowledge of available resources. This implies worst-case resource allocation (implementations may not use any more resources than declared), and that resources may not be used by processes outside of this specification.

Planning for deployment can happen “online” or “offline.” In the online case, the planner considers the presently available resources that are returned from **getAvailableResources**. In offline planning, the planner considers all available resources in order to plan for an application that is to be deployed into an “empty” target environment.

It may be necessary to serialize access to resource information and planning using means beyond the scope of this specification, in order to avoid race conditions in online planning – otherwise resources might be committed elsewhere while planning, or multiple plans might end up competing for the same resources.

2.6.2 DomainUpdateKind

Description



The <ClassName>DomainUpdateKind is an enumeration used as a parameter to the updateDomain operation of the <ClassName>TargetManager to describe how <ClassName>Domain information is to be updated.

Attributes

No attributes.

Associations

No associations.

Constraints

No constraints.

Semantics

If the `Add` kind is used, then information about nodes, interconnects, bridges and shared resources is added to the <ClassName>Domain. In case of **Delete**, information is removed. In case of **UpdateAll**, existing information about the full capacity of resources is updated. In case of **UpdateAvailable**, information about the available capacity of resources is updated.

2.7 Execution Data Model

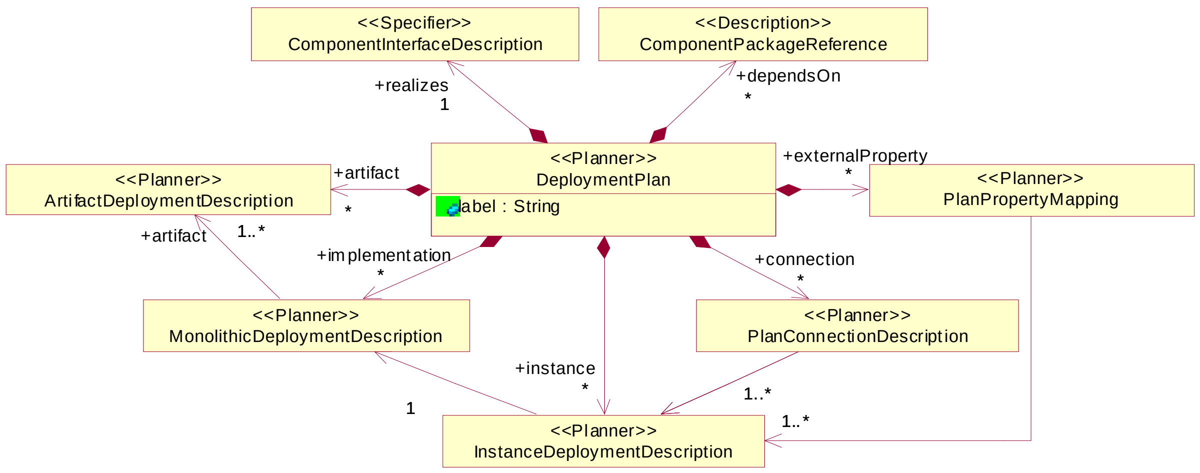
The following classes are part of the Execution Data Model. They are placed in the Execution subpackage of the Deployment and Configuration package.

Before deployment can occur, decisions must be made about the implementations to select (if multiple implementations exist in a package) and where to deploy each monolithic component implementation. All information about an application's deployment is collected in a <ClassName>DeploymentPlan. This plan can be used transiently (i.e., executed right away), or it may be stored to avoid the overhead of planning in the future. The <ClassName>DeploymentPlan can be used by an <ClassName>ExecutionManager to create a specific factory object for the application. A <ClassName>DeploymentPlan is "standalone" in that it does not necessarily refer to a repository, only to artifacts, which, depending on the implementation, may or may not reside in the repository.

Details about each class in the Execution Data Model will be presented in the following sections.

2.7.1 DeploymentPlan

Description



The <ClassName>DeploymentPlan contains information about artifacts that are part of the deployment (<ClassName>ArtifactDeploymentDescription), how to create component instances from artifacts (<ClassName>MonolithicDeploymentDescription), and where to instantiate them (<ClassName>InstanceDeploymentDescription). It then contains information about connections between them (<ClassName>AssemblyConnectionDescription) and about the mapping of external properties. It finally contains information about the component interface that is realized by the application. The <ClassName>DeploymentPlan is analogous to the **ComponentAssemblyDescription** in the Component Data Model. In fact, the <ClassName>DeploymentPlan can be seen as a flattened assembly (without recursion). In the plan, all assemblies have been recursively replaced by their white-box representation, and concrete implementations have been chosen for each subcomponent. All that remains are the leaf nodes, i.e. components that have a monolithic implementation.

To avoid redundancy, a Planner can compare the identity of artifacts and component implementations for identity (using their **UUID** attributes) and then share <ClassName>ArtifactDeploymentDescription and <ClassName>MonolithicDeploymentDescription elements.

Attributes

label: String	Users may optionally assign a human readable label to a <ClassName>DeploymentPlan.

Associations

artifact: <ClassName>ArtifactDeploymentDescription [*]
Implementation artifacts related to the deployment.

implementation: <ClassName>MonolithicDeploymentDescription
Component implementations used in the deployment.

instance: <ClassName>InstanceDeploymentDescription [*]
Component instances that are to be created.

connection: <ClassName>PlanConnectionDescription [*]
Connections that are to be made between the component instances, the application's external ports, or external locations.

externalProperty: <ClassName>PlanPropertyMapping [*]
Maps the application's external properties to properties of component instances.

realizes: <ClassName>ComponentInterfaceDescription [1]
The component interface implemented by the application.

dependsOn: **ComponentPackageReference** [*]
Implementations of these interfaces must be executing in the target environment before deploying this plan is possible. Copied from the **ComponentImplementationDescription** element.

Constraints

No constraints.

Semantics

The <ClassName>DeploymentPlan is a self-contained piece of information that contains all necessary data about the deployment of an application to a specific target environment.

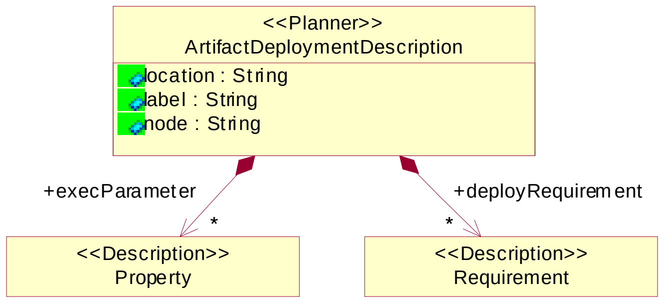
The deployment engine that is part of the <ClassName>ExecutionManager or <ClassName>ApplicationManager traverses the instances; for each instance, it determines the implementation and its artifacts, which need to be installed on a target node prior to component instantiation. All artifacts used in this process are marked. The deployment engine then traverses the artifacts and processes all "leftover" <ClassName>ArtifactDeploymentDescription elements; these may be additional artifacts included by the Planner to take care of special conditions in the target environment.

The deployment engine then proceeds to create the component instances and interconnects them.

The interface information is used so that the application can present this interface to the user. (This is detailed by platform specific models.) Default values for properties (the **configProperty** elements of the <ClassName>ComponentInterfaceDescription) are not needed in the plan and ignored by the deployment engine; a Planner may decide not to copy them into the plan.

2.7.2 ArtifactDeploymentDescription

Description



<ClassName>ArtifactDeploymentDescription describes an artifact that is to be deployed as part of the plan. It mirrors the <ClassName>ImplementationArtifactDescription from the component data model. To avoid redundancy, this element can be shared among <ClassName>InstanceDeploymentDescription elements, should component instances use the same artifact more than once. A Planner can compare artifacts for identity using the **UUID** attribute of the <ClassName>ImplementationArtifactDescription element. <ClassName>ArtifactDeploymentDescription describes the installation of a single implementation artifact on a node as part of component instantiation. It contains an URL pointing to the <ClassName>ImplementationArtifact. Execution parameters and deployment requirements are copied from the <ClassName>ImplementationArtifactDescription.

Attributes

location: **String**	The location where the artifact can be loaded from. Copied from the <ClassName>ImplementationArtifactDescription.
label: **String**	An optional human readable label identifying the artifact.
node: **String**	The name of the node where the artifact is to be installed. If blank, the node is implied by the <ClassName>InstanceDeploymentDe-scription parent.

Associations

execParameter: <ClassName>Property [*]	Execution parameters, copied from the <Class-Name>ImplementationArtifactDescription.
deployRequirement: <ClassName>Requirement [*]	Deployment requirements, copied from the <ClassName>Imple-mentationArtifactDescription.

Constraints

No constraints.

Semantics

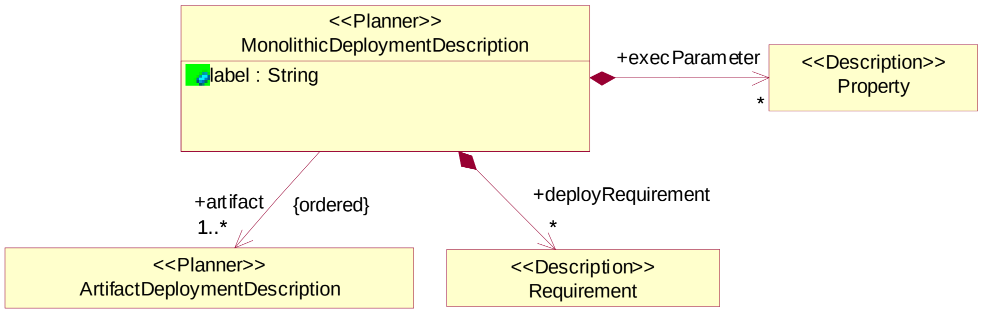
The deployment requirements carry information about the resources used by this implementation artifact, so that they can be committed by the <ClassName>TargetManager (presumably via the <ClassName>ExecutionManager).

Usually, the `node` attribute is the empty string, so that artifacts will be deployed on the node where a component is to be instantiated as implied by the <ClassName>InstanceDeploymentDescription. The attributed is included here for the exotic case that special artifacts need to be installed in the target environment. In that case, the Planner would add <ClassName>ArtifactDeploymentDescription elements to the plan that are unrelated to component instances.

A Planner can generate a human readable `label` attribute from the `label` attributes of packages, implementations, assembly subcomponents and <ClassName>ImplementationArtifactDescription elements, i.e. a "path" describing the origin of this artifact. In case of an error, a user can then use this label to identify the source of a problem.

2.7.3 MonolithicDeploymentDescription

Description



<ClassName>MonolithicDeploymentDescription describes the deployment of a component as part of the plan. It mirrors the <ClassName>MonolithicImplementationDescription from the component data model. If the same component instance is deployed more than once, a <ClassName>MonolithicDeploymentDescription can be shared by multiple <ClassName>InstanceDeploymentDescription elements. A Planner can compare monolithic implementations for identity using the **UUID** attribute of the **ComponentImplementationDescription**. The <ClassName>MonolithicDeploymentDescription contains a human-readable label and references <ClassName>ArtifactDeploymentDescription elements for all artifacts that are part of the deployment. The execution parameters and deployment requirements are copied from the <ClassName>MonolithicImplementationDescription.

Attributes

label: **String**	An optional human readable label identifying the component implementation.

Associations

artifact: <ClassName>ArtifactDeploymentDescription [*]	The implementation artifacts that are part of this monolithic component implementation.
execParameter: <ClassName>Property [*]	Execution parameters, copied from the <ClassName>MonolithicImplementationDescription.
deployRequirement: <ClassName>Requirement [*]	Deployment requirements, copied from the <ClassName>MonolithicImplementationDescription.

Constraints

No constraints.

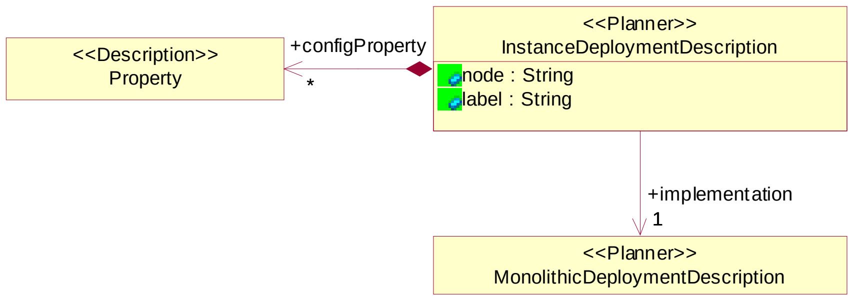
Semantics

The artifacts referenced here represent a depth-first traversal of the primary artifacts from the <ClassName>MonolithicImplementationDescription and their dependency. A depth-first traversal ensures that all dependees can be installed before the dependent artifacts.

A Planner can generate a human readable `label` from the `label` attributes of packages, implementations, assembly subcomponents and **ComponentImplementationDescription** elements, i.e. a “path” describing the origin of this component implementation. In case of an error, a user can then use this label to identify the source of a problem.

2.7.4 InstanceDeploymentDescription

Description



<ClassName>InstanceDeploymentDescription contains the information that is necessary in order to deploy a single component instance. It references a <ClassName>MonolithicDeploymentDescription and includes the name of the node where the component is to be instantiated. It then contains properties that are used to configure the component instance.

Attributes

node: **String**	The name of the node where the component is to be instantiated.
label: **String**	A human readable label.

Associations

implementation: <ClassName>MonolithicDeploymentDescription [1]
The component that is to be instantiated.
configProperty: <ClassName>Property [*] Properties to configure the component instance after instantiation.

Constraints

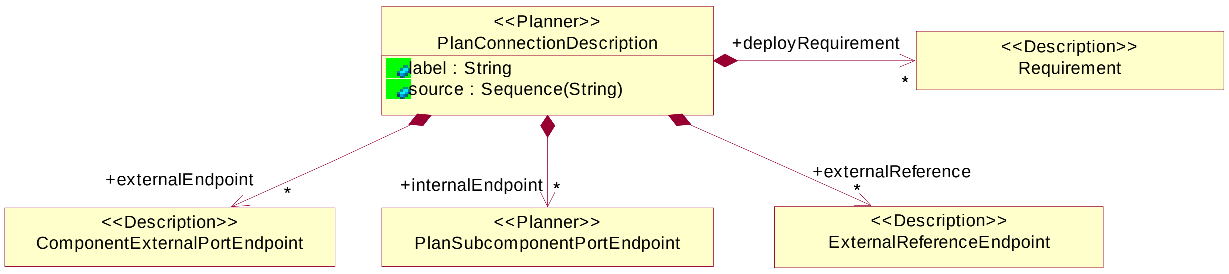
No constraints.

Semantics

A Planner can generate a human readable `label` from the `label` attributes of packages, assembly implementations and assembly subcomponents, i.e. a "path" describing the origin of this component instance. In case of an error, a user can then use this label to identify the source of a problem.

2.7.5 PlanConnectionDescription

Description



The <ClassName>PlanConnectionDescription describes a connection that is to be made among ports within the application that is being deployed. It is analogous to the <ClassName>AssemblyConnectionDescription that describes a connection within an assembly. The <ClassName>ComponentExternalPortEndpoint and <ClassName>ExternalReferenceEndpoint elements are reused from the Component Data Model.

Attributes

label: **String**	A label that uniquely identifies this element of the <ClassName>DeploymentPlan.
source: **Sequence(String)**	The labels of all <ClassName>AssemblyConnectionDescription elements that were combined into this <ClassName>PlanConnectionDescription.

Associations

`deployRequirement`: <ClassName>Requirement [*]
Connection requirements; the sum of all deployment requirements of all <ClassName>AssemblyConnectionDescription elements that are involved in this connection.

externalEndpoint: <ClassName>ComponentExternalPortEndpoint [*]
Identifies a port of the component that is implemented by the application as an endpoint of this connection.

internalEndpoint: <ClassName>PlanSubcomponentPortEndpoint [*]
Identifies a port of a component within the application as an endpoint of this connection.

externalReference: <ClassName>ExternalReferenceEndpoint [*]
Identifies a location outside the application as an endpoint of this connection.

Constraints

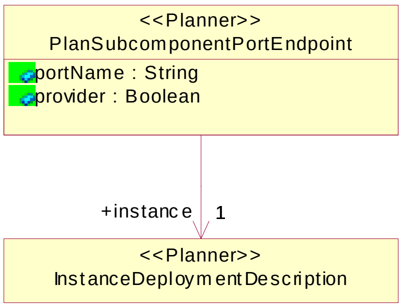
The number of endpoints must be larger than one.

Semantics

During application launch, a connection between all endpoints will be established.

2.7.6 PlanSubcomponentPortEndpoint

Description



Identifies a port of a component within the application as an endpoint of the connection described by the <ClassName>PlanConnectionDescription that this element is contained in.

Attributes

portName: **String**	The name of the port of the associated component instance that is to be an endpoint of this connection.
provider: **String**	Identifies whether the port is a provider or user port.

Associations

instance: <ClassName>InstanceDeploymentDescription [1]
The associated component instance.

Constraints

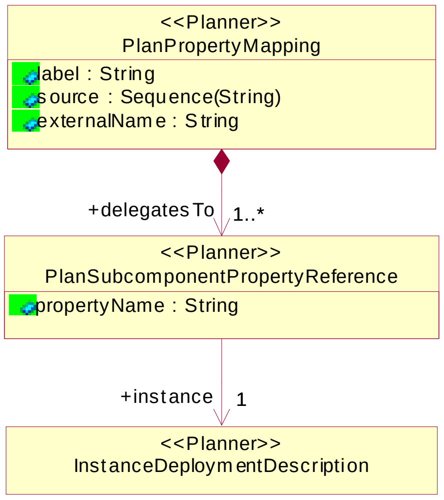
The port name must be valid for the referenced component.

Semantics

See above.

2.7.7 PlanPropertyMapping

Description



<ClassName>PlanPropertyMapping is part of the <ClassName>DeploymentPlan. It identifies a property of the component that this application is implementing and the subcomponents' properties that it delegates to.

Attributes

label: **String**	A label that uniquely identifies this element of the <ClassName>DeploymentPlan.
source: **Sequence(String)**	The labels of all <ClassName>AssemblyPropertyMapping elements that were combined into this <ClassName>PlanPropertyMapping.
externalName: **String**	The name of a property of the component that the application is implementing.

Associations

delegatesTo: <ClassName>PlanSubcomponentPropertyReference [1..*]
References ports of subcomponents within the application that the property is delegated (or propagated) to.

Constraints

The **externalName** must match the name of a property of the component that the assembly is implementing.

2.7.8 PlanSubcomponentPropertyReference

Description

Identifies a property of a subcomponent within the deployment plan that an external property of the component that the application implements delegates to.

Attributes

propertyName: **String**	The name of the property of the associated component instance that the external property is delegated to.

Associations

instance: <ClassName>InstanceDeploymentDescription [1]
The associated component instance.

Constraints

The `propertyName` must match the name of a property of the associated component.

Semantics

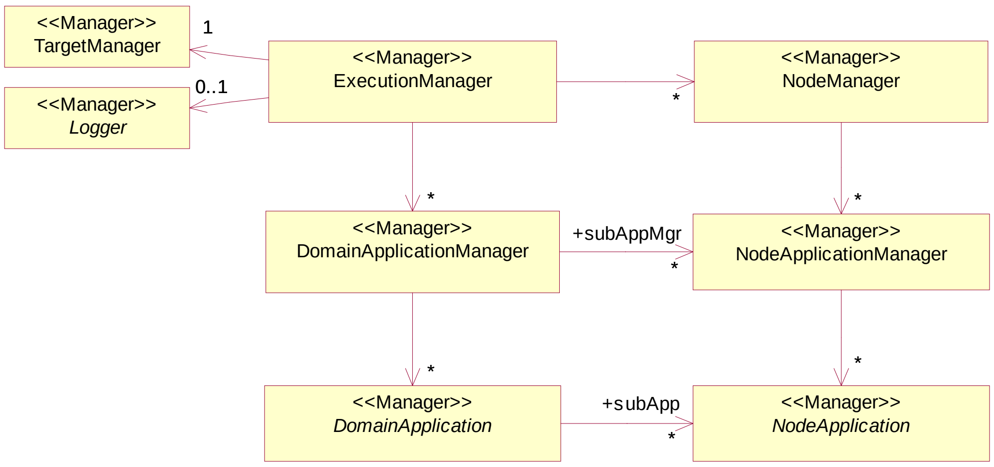
No semantics.

2.8 Execution Management Model

The following classes are part of the Execution Management Model. They are placed in the Execution subpackage of the Deployment and Configuration package.

Application	**Page 58**
ApplicationManager	**Page 55**
Connection	**Page 61**
DomainApplication	**Page 59**
DomainApplicationManager	**Page 56**
Endpoint	**Page 61**
ExecutionManager	**Page 51**
Logger	**Page 60**
NodeApplication	**Page 60**
NodeApplicationManager	**Page 57**
NodeManager	**Page 53**

2.8.1 Execution Management Model Overview



After planning, application execution happens in two phases, in a total of three steps. The first phase is the preparation of the plan for execution using the `preparePlan` operation of the <ClassName>ExecutionManager, resulting in an <ClassName>ApplicationManager factory object, which can be used to put the plan into action, potentially more than once. The second phase, launching the application, is divided into two steps. The first step of launching is calling the `startLaunch` operation on the <ClassName>ApplicationManager. This causes the <ClassName>Application to be executed, but not to be started yet. The second step of launching is calling the `finishLaunch` operation on the <ClassName>Application. The reason for splitting application launch into two steps is launch-time configuration and interconnection. The first step returns references to ports that are provided by the application, the second step supplies references to ports that are used by the application.

Application execution involves the “domain” level and the “node” level. On the domain level, the <ClassName>ExecutionManager manages the execution of an application into the domain. The <ClassName>ExecutionManager separates the “global” application into “local” sub-applications that execute within a node. This essentially creates “virtual components” to run entirely within a node, including intra-node connections. The deployment of virtual components onto a node can be described the same way as the deployment of the original application, using a <ClassName>DeploymentPlan, with the limitation that all component instances will be located on the same node.

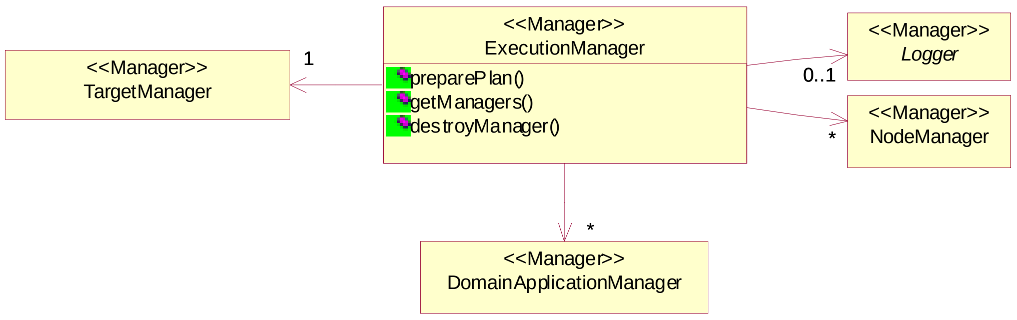
The <ClassName>ExecutionManager creates deployment plans for virtual components to run on each node, so that the complete application is covered. It then passes each <ClassName>DeploymentPlan to the <ClassName>NodeManager that is responsible for instantiating components on that node.

Just as the <ClassName>DeploymentPlan structure is the same for the deployment of both the global application and the local applications, the interfaces for managing them, <ClassName>ApplicationManager and <ClassName>Application, are the same. To keep the semantics separate, global and local versions of both interfaces are introduced with the Domain and Node prefixes. During launch and shutdown, global <ClassName>DomainApplicationManager and <ClassName>DomainApplication instances delegate management to the local, node-specific <ClassName>NodeApplicationManager and <ClassName>NodeApplication managers with the same interface.

The separation between <ClassName>ExecutionManager and <ClassName>NodeManager serves the purpose of creating a vendor boundary. It uncouples deployment (implemented by the vendor of the deployment engine) from the execution of components (implemented by the vendor of the hardware or development environment). This allows hardware vendors to supply a node-specific <ClassName>NodeManager, <ClassName>NodeApplicationManager and <ClassName>NodeApplication implementations that can then interact with any deployment engine.

2.8.2 ExecutionManager

Description



The <ClassName>ExecutionManager manages the execution of applications from a <ClassName>DeploymentPlan. It has knowledge of <ClassName>NodeManager instances that manage nodes within the domain, and will delegate execution of component instances to relevant <ClassName>NodeManager instances as described by the plan. The <ClassName>ExecutionManager is also associated with a <ClassName>TargetManager for resource management, and, optionally, a centralized logging facility.

Application execution is initiated by preparing a <ClassName>DeploymentPlan using the **preparePlan** operation. This creates a new <ClassName>DomainApplicationManager that can later be used to launch one or more application instances.

Operations

preparePlan (**plan**: <ClassName>DeploymentPlan, **commitResources**: **Boolean**): <ClassName>DomainApplicationManager

Creates an application manager (factory) from a deployment plan. If **commitResources** is true, then resources used by the plan will be committed. If false, then it is assumed that resources were already committed by an online planner. Raises the <ClassName>ResourceNotAvailable exception if **commitResources** is true, if early resource allocation is used, and one of the requested resources is not available. Raises the <ClassName>StartError exception if a deployment-related error occurs during preparation. Raises the <ClassName>PlanError exception if there is a problem with the plan.

destroyManager (**manager**: <ClassName>DomainApplicationManager)

Terminates an application manager and free all associated resources. All running applications are terminated as well. Raises the <ClassName>StopError exception if a problem occurs terminating or unpreparing any application. Raises the <ClassName>InvalidReference exception if the manager is unknown.

getManagers (): **Sequence(<ClassName>DomainApplicationManager)**

Returns a list of all active application managers.

Associations

domainApplicationManager: <ClassName>DomainApplicationManager [*]

An <ClassName>ExecutionManager instantiates <ClassName>DomainApplicationManager instances.

targetManager: <ClassName>TargetManager [1]

The <ClassName>TargetManager that will be used for resource commitments.

logger: <ClassName>Logger [0..1] An optional logging faciltiy.

nodeManager: <ClassName>NodeManager [*] <ClassName>NodeManager references for all nodes that are part of the domain.

Constraints

No constraints.

Semantics

The semantics of preparation are undefined. Preparation usually involves the distribution of artifacts to the nodes. However, implementations might decide to delay this distribution until application launch — or they might, on the other hand, preload artifacts into memory so that launch can happen as fast as possible.

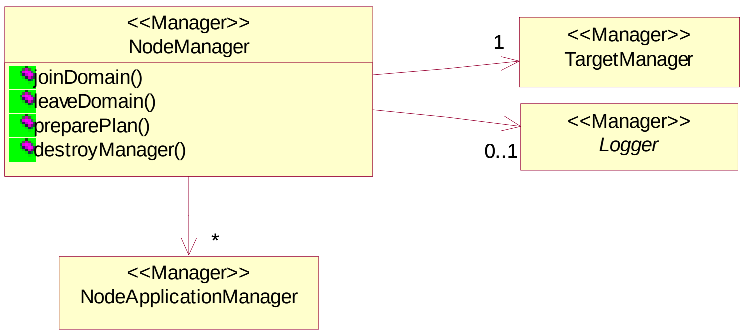
It is also undefined whether resource commitment (in case the **commitResources** parameter to the **preparePlan** operation is true) happens at preparation or launch time. Implementations should document their behavior in this respect.

The **preparePlan** operation takes the deployment plan and prepares “virtual components” with the subset of the application that is to be executed on each node. The <ClassName>ExecutionManager then contacts the <ClassName>NodeManager instances that are responsible for each node, and passes their piece of the application to their **preparePlan** operation, using the same <ClassName>DeploymentPlan format. This results in a “global” level <ClassName>DomainApplicationManager that holds references to “local,” node-specific <ClassName>NodeApplicationManager instances for each piece of the application.

The `destroyManager` operation releases all resources that were allocated during preparation and launch.

2.8.3 NodeManager

Description



The <ClassName>NodeManager is responsible for managing a partial applications that is limited to its node. It mirrors the <ClassName>ExecutionManager, but is limited to one node only.

Operations

joinDomain (**domainSubset**: <ClassName>Domain, `manager`: <ClassName>TargetManager, `log`: <ClassName>Logger)
Informs the <ClassName>NodeManager that it is now part of a <ClassName>Domain. The **domainSubset** contains the resource availability information that is currently known within the domain. **manager** is a reference to the <ClassName>TargetManager to (optionally) send domain updates to. log is an abstract (PSM defined) class to send log messages to.

leaveDomain ()
Informs the <ClassName>NodeManager that it is being removed from the domain, e.g. because of domain shutdown.

preparePlan (**plan**: <ClassName>DeploymentPlan): <ClassName>NodeApplicationManager
Prepares a partial application. The part of the application that is to be executed on this node is expressed as a <ClassName>DeploymentPlan that implements a “virtual component” with the subcomponents, connections, external ports and properties. Raises the <ClassName>StartError exception if a deployment-related error occurs during preparation. Raises the <ClassName>PlanError exception if there is a problem with the plan.

destroyManager (**manager**: <ClassName>NodeApplicationManager)
Terminates a <ClassName>NodeApplicationManager and frees all associated resources. All running applications are terminated. Rais-

es the <ClassName>StopError exception if an error occurs during termination. Raises the <ClassName>InvalidReference exception if the manager reference is unknown.

Associations

targetManager: <ClassName>TargetManager [1]
The <ClassName>TargetManager that <ClassName>Domain updates are sent to if necessary. This is the reference passed as a parameter to the `joinDomain` operation.

logger: <ClassName>Logger [0..1] The <ClassName>Logger to send log messages to. If the <ClassName>NodeManager wants to produce log messages, it keeps the reference passed as a parameter to the `joinDomain` operation.

nodeApplicationManager: <ClassName>NodeApplicationManager [*]
The node-specific application managers instantiated by this <ClassName>NodeManager via the `preparePlan` operation.

Constraints

No constraints.

Semantics

The **joinDomain** operation is called by the <ClassName>ExecutionManager at startup time or when it is informed of a new node via the **updateDomain** operation. Both the **joinDomain** and **leaveDomain** operations are called by the <ClassName>ExecutionManager on user request to add or remove nodes from a domain.

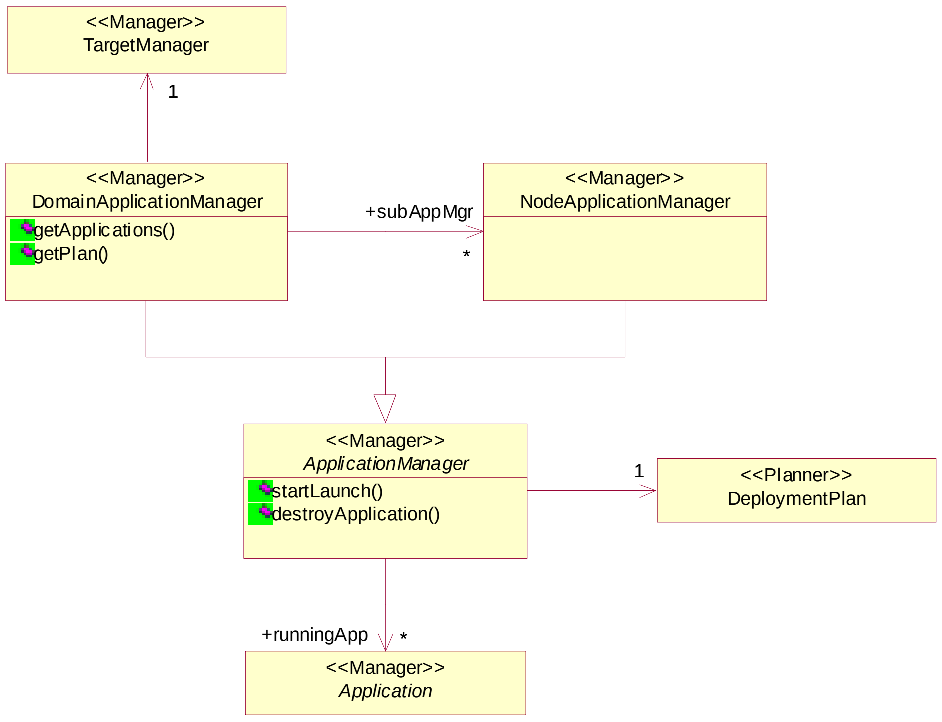
If the **joinDomain** operation is called, the <ClassName>NodeManager may optionally examine the `domainSubset`, and send an update message to the <ClassName>TargetManager if discrepancies are found.

The semantics of the **leaveDomain** operation are undefined. A <ClassName>NodeManager might shutdown or reset. In particular, the effect on running applications is also undefined. Behavior of a <ClassName>NodeManager implementation should be well documented. A <ClassName>NodeManager should not log any messages after returning from the **leaveDomain** operation.

The **preparePlan** operation and **destroyApplication** operations are called by the <ClassName>ExecutionManager as a result of a user demand for application preparation or destruction. The <ClassName>DeploymentPlan that is passed to the **preparePlan** operation describes a virtual component that is composed of all subcomponents and connections that are to be made within the node, plus mappings for connections and properties that external to that node.

2.8.4 ApplicationManager

Description



An <ClassName>ApplicationManager is used to first launch and later to terminate an application according to a concrete <ClassName>DeploymentPlan. <ClassName>ApplicationManager is an abstract class that is specialized by the <ClassName>DomainApplicationManager, which handles deployment of a "global" application, and the <ClassName>NodeApplicationManager, which handles deployment of a locality constrained application onto a single node.

Operations

startLaunch (**configProperty**: **Sequence(Property)**,
out providedReference: **Sequence(<ClassName>Connection)**):
<ClassName>Application
Executes the application, but does not start it yet. Users can optionally provide launch-time configuration properties to override properties that are part of the plan. A handle to the application is returned, as well as connections for the component's external provider ports. Raises the <ClassName>InvalidProperty exception if a configuration property is invalid. Raises the <ClassName>StartError exception if an error occurs during launching. Raises the <ClassName>ResourceNotAvailable exception if the

commitResources parameter to the prepare operation of the <ClassName>ExecutionManager was true, if late resource allocation is used, and one of the requested resources is not available.

destroyApplication (**app**: <ClassName>Application)
Terminates a running application. Raises the <ClassName>StopError exception if an error occurs during termination. Raises the <ClassName>InvalidReference exception if the appliction reference is unknown.

Associations

runningApp: <ClassName>Application [*]The applications that were launched but not terminated yet.

deploymentPlan: <ClassName>DeploymentPlan [1]
The <ClassName>DeploymentPlan that this <ClassName>ApplicationManager is based on, a copy of the plan that was passed to the **preparePlan** operation of the <ClassName>ExecutionManager or <ClassName>NodeManager.

targetManager: <ClassName>TargetManager [1]
The <ClassName>TargetManager that is used to commit resources if necessary.

Constraints

Depending on the plan and whether it was based on static or online resource data, launching multiple applications from the same <ClassName>ApplicationManager in parallel might fail because of resource constraints.

Semantics

The behavior of an <ClassName>ApplicationManager is different depending on whether it is used as a <ClassName>DomainApplicationManager on the “global” level (if instantiated from an <ClassName>ExecutionManager) or a <ClassName>NodeApplicationManager on the “local” level (if instantiated from a <ClassName>NodeManager). Implementations for these two cases are usually separate. An <ClassName>ExecutionManager implementation has access to <ClassName>DomainApplicationManager and <ClassName>DomainApplication implementations, a <ClassName>NodeManager has access to <ClassName>NodeApplicationManager and <ClassName>NodeApplication implementations.

2.8.5 DomainApplicationManager

Description

The <ClassName>DomainApplicationManager is responsible for deploying an application on the domain level, i.e. across nodes. It specializes the <ClassName>ApplicationManager interface.

Operations

getApplications (): **Sequence (<ClassName>Application)**
Returns a list of all applications that have been launched from this <ClassName>ApplicationManager and that are still executing.

getPlan (): <ClassName>DeploymentPlan Returns the <ClassName>DeploymentPlan associated with this <ClassName>ApplicationManager.

Associations

subAppMgr: <ClassName>NodeApplicationManager [*]
The manager for the pieces of the application that run on each node.

targetManager: <ClassName>TargetManager [1]
The <ClassName>TargetManager that is used to commit resources if necessary.

Constraints

The targets of the `runingApp` association (inherited from <ClassName>ApplicationManager) are instances of <ClassName>DomainApplication.

Semantics

A <ClassName>DomainApplicationManager has references to node-specific <ClassName>NodeApplicationManager elements as created by the **preparePlan** operation of the <ClassName>ExecutionManager. The **startLaunch** operation then calls **startLaunch** on the <ClassName>NodeApplicationManager instances, passing the relevant properties and collecting the returned connections as determined by the separation of the “global” <ClassName>DeploymentPlan into node-specific plans. The same applies to the **destroyApplication** operation.

2.8.6 NodeApplicationManager

Description

The <ClassName>NodeApplicationManager is responsible for deploying an locality constrained application onto a node. It specializes the <ClassName>ApplicationManager interface.

Operations

No additional operations.

Associations

No additional associations.

Constraints

The targets of the **runingApp** association (inherited from <ClassName>ApplicationManager) are instances of <ClassName>NodeApplication.

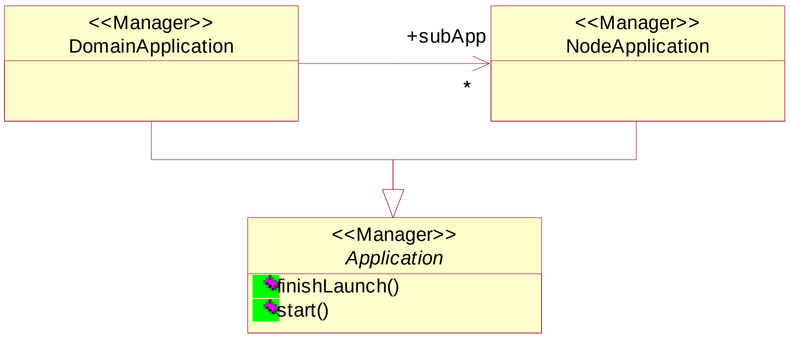
The associated <ClassName>DeploymentPlan (inherited from <ClassName>ApplicationManager) only contains instance deployments onto the node that is represented by the <ClassName>NodeManager parent.

Semantics

A <ClassName>NodeApplicationManager is responsible for executing and terminating component instances on the node that it is part of (as defined by the <ClassName>NodeManager parent, usually but not necessarily implying co-location).

2.8.7 Application

Description



<ClassName>Application is an abstract class represents a running application. The <ClassName>Application class may be mapped to different classes in a platform specific models, potentially allowing navigation to an application's ports, configuration or introspection at runtime. <ClassName>Application is specialized by <ClassName>DomainApplication, which represents a "global" application (i.e. across nodes), and <ClassName>NodeApplication, which represents a locality constrained application that is running on a single node.

Operations

finishLaunch (**providedReference**: **Sequence(<ClassName>Connection)**, **start**: **Boolean**)

The second step in launching an application. External references may be provided to connect to the component's external user ports. If the start parameter is true, the application is started as well. Raises the <ClassName>InvalidConnection if one of the provided references is invalid. Raises the <ClassName>StartError exception if launching or starting the application fails.

start ()

Starts the application. Raises the <ClassName>StartError exception if starting the application fails.

Associations

No associations.

Constraints

No constraints.

Semantics

The **finishLaunch** operation must be called in order to complete the component's configuration.

If clients want to start multiple applications simultaneously, they can set the start parameter of the **finishLaunch** operation to false and then call the **start** operation separately. If clients want to avoid the additional round-trip, they can set the start parameter of the **finishLaunch** operation to true; in that case, the start operation needs not be called.

The behavior of a <ClassName>Application is different depending on whether it is used on a "global" level (if its parent is a <ClassName>DomainApplicationManager) or on a "local" level (if its parent is a <ClassName>NodeApplicationManager). Implementations for these two cases are usually separate. A <ClassName>DomainApplicationManager only creates <ClassName>DomainApplication instances, a <ClassName>NodeApplicationManager only creates <ClassName>NodeApplication instances.

A node-specific <ClassName>Application represents running component instances on the node that it is part of (as defined by the <ClassName>NodeManager parent, usually but not necessarily implying co-location).

2.8.8 DomainApplication

Description

A <ClassName>DomainApplication represents a "global" application that was deployed across nodes. It has the same interface as <ClassName>Application, but has different semantics.

Operations

No additional operations.

Associations

subApp: <ClassName>NodeApplication [*]The pieces of the application that run on each node.

Constraints

No constraints.

Semantics

A “global” <ClassName>DomainApplication has references to node-specific <ClassName>NodeApplication elements as created by the **startLaunch** operation of the <ClassName>DomainApplicationManager. The **finishLaunch** operation then calls **finishLaunch** on the node-specific <ClassName>NodeApplication instances, passing the relevant connections as determined by the separation of the “global” <ClassName>DeploymentPlan into node-specific plans. The same applies to the **destroyApplication** operation.

2.8.9 NodeApplication

Description

<ClassName>NodeApplication represents a piece of an application that is executing within a single domain.

Operations

No additional operations.

Associations

No additional associations.

Constraints

No constraints.

Semantics

<ClassName>NodeApplication has the same semantics as the <ClassName>Application base class. It interconnects and starts the piece of the application that is being launched on the node that is represented by the <ClassName>NodeManager parent.

2.8.10 Logger

Operations

No operations.

Associations

No associations.

Constraints

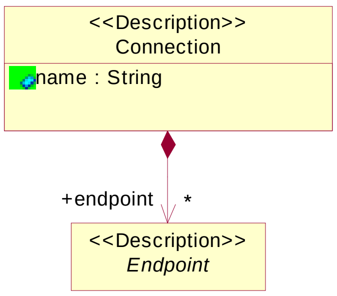
No constraints.

Semantics

<ClassName>Logger is an abstract runtime class to facilitate logging within the domain. It has to be mapped to a concrete type by platform specific models.

2.8.11 Connection

Description



A <ClassName>Connection is used to describe connections from or to a component port at runtime.

Attributes

name: **String**	The name of the component's port.

Associations

endpoint: <ClassName>Endpoint [*]The endpoints that are part of the connection.

Constraints

No constraints.

Semantics

No additional semantics.

2.8.12 Endpoint

Attributes

No attributes.

Associations

No associations.

Constraints

No constraints.

Semantics

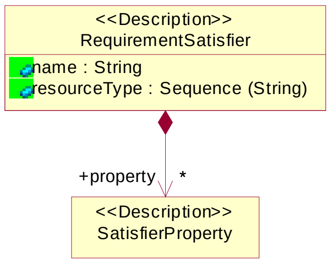
<ClassName>Endpoint is an abstract class that contains the “address” of an endpoint. This class needs to be mapped into a concrete platform specific type.

2.9 Common Elements

This section contains common model elements that are shared between multiple segments. They are placed in the Common subpackage of the Deployment and Configuration package.

2.9.1 RequirementSatisfier

Description



<ClassName>RequirementSatisfier describes a resource or capability that can satisfy a requirement.

Attributes

name: **String**	An optional name for the requirement satisfier.
resourceType: **Sequence(String)**	The resource types that can be satisfied by this satisfier.

Associations

property: <ClassName>SatisfierProperty [*]	Properties associated with this satisfier.

Constraints

There must be at least one element in the **resourceType** sequence attribute.

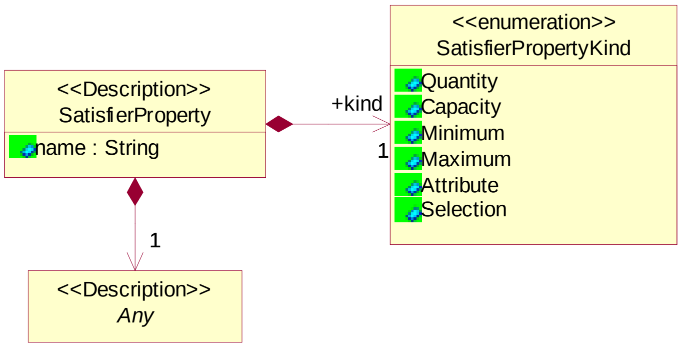
context RequirementSatisfier inv:
self.resourceType->size() >= 1

Semantics

The type of a <ClassName>Requirement is must match one of the elements in the **resourceType** attribute. The requirement's properties will then be matched against the satisfier's properties.

2.9.2 SatisfierProperty

Description



Describes a specific property of a <ClassName>Resource or <ClassName>SharedResource. It contains a <ClassName>SatisfierPropertyKind that classifies the <ClassName>SatisfierProperty and has implications on the type of the value and the comparison between the <ClassName>SatisfierProperty and a required <ClassName>Property.

Attributes

name: String	The name of the property.

Associations

kind: <ClassName>SatisfierPropertyKind [*]	The kind of the property.
value: <ClassName>Any [1]	The value of the property.

Semantics

<ClassName>SatisfierProperty elements are matched against the <ClassName>Property elements within a <ClassName>Requirement at planning time. They describe attributes and capacities of hardware or software. The `name` attribute of the <ClassName>SatisfierProperty must match the `name` attribute of the <ClassName>Property it is compared against. Matching the values will be discussed as part of the <ClassName>SatisfierPropertyKind semantics. The type of the value may be fully or partially implied by the kind.

2.9.3 SatisfierPropertyKind

Description

Classifies a <ClassName>SatisfierProperty. Each <ClassName>SatisfierPropertyKind identifies a specific way to match requirements against resources. The kind of <ClassName>SatisfierPropertyKind implies the types of the values contained in <ClassName>SatisfierProperty and <ClassName>Property, and the algorithm to check their compatibility.

Attributes

No attributes.

Associations

No associations.

Semantics

The value of this enumeration implies how to check for compatibility between a required property and a resource's property, and how to keep track of capacities. In the following text, "property" refers to the property element of the <ClassName>SatisfierProperty, and "requirement" refers to the property element of the <ClassName>Requirement. Both must have matching names.

Quantity	This property exists in a certain quantity, but its capacity is not considered. The value of the property is of integer type. The value of the requirement is ignored, but each time this property is used, the quantity is decreased by one until zero. To match the requirement, the property must have a value of at least one. Example: a sound card with 4 output channels.
Capacity	This property has a certain capacity that can be consumed. The value of the property and the requirement property are both of numerical type. The value of the requirement is subtracted from the value of the property. To match the requirement, the property must have a value that equals or exceeds the value of the requirement. Example: memory size.
Minimum	The property describes a capability with a lower bound. The value of the property and the requirement are both of a type that supports ordering. To match, the value of the requirement must equal or exceed the value of the property. Example: latency – e.g. the resource can guarantee 30ms latency, the property requires at least 40ms.
Maximum	The property describes a capability with an upper bound. The value of the property and the requirement are both of a type that supports ordering. To match, the value of the requirement must be equal or lesser than the value of the property. Example: CPU speed – e.g. the property has 700MHz, and there is a requirement on at least 500MHz.

Attribute	The value of the property and the requirement are both of a type that supports equality comparison. To match, the requirement must compare equal to the property. Example: OS type.
Selection	The type of the property is a sequence of a type that supports equality comparison, the requirement is a single value of the same type. To match, the value of the requirement must compare equal to one element of the property values.

Platforms have to specify concrete types to be used for the comparison of the **Minimum**, **Maximum**, **Attribute** and **Selection** kinds, and define how to order and compare them.

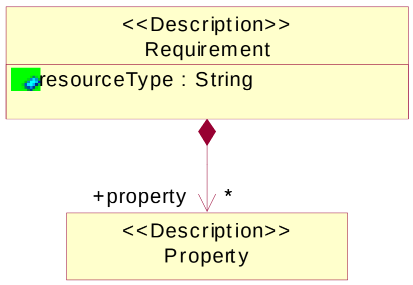
Domains have to define resource types, their properties, and the kinds to use for each property.

The **Quantity** and **Attribute** kinds are redundant, but included here to account for these common use cases. (**Quantity** is equivalent to a **Capacity** that is required in amounts of one, and **Attribute** is a subset of **Selection**.)

The above list of resource kinds is expected to cover the most common use cases. Platform specific models and domain specific profiles are allowed to add more kinds if necessary.

2.9.4 Requirement

Description



<ClassName>Requirement is used in the <ClassName>MonolithicImplementationDescription, <ClassName>ImplementationArtifactDescription and the <ClassName>AssemblyConnectionDescription to express that the implementation artifact or connection has requirements that must be fulfilled by resources in the target environment. The resource type must match the type of a resource.

Attributes

resourceType: String	Identifies the resource type.

Associations

properties: <ClassName>Property [*]	Properties associated with the resource.

Constraints

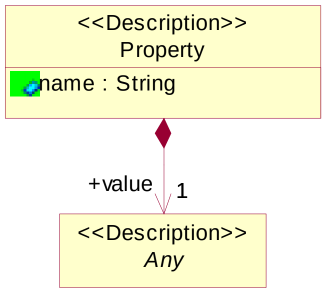
No constraints.

Semantics

No semantics.

2.9.5 Property

Description



A <ClassName>Property has a name and a value. It is used to carry named and values in various places.

Attributes

name: **String**	The name of the property.

Associations

value: <ClassName>Any [1]	Contains the value.

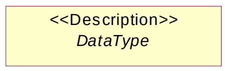
Constraints

No constraints.

Semantics

No semantics.

2.9.6 DataType



Attributes

No attributes.

Associations

No associations.

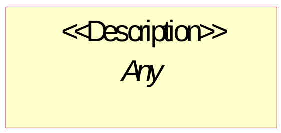
Constraints

No constraints.

Semantics

<ClassName>DataType is an abstract class that describes a data type. This class needs to be mapped into a concrete platform specific type.

2.9.7 Any



Attributes

No attributes.

Associations

No associations.

Constraints

No constraints.

Semantics

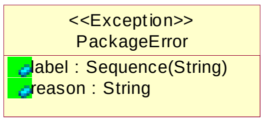
<ClassName>Any is an abstract class that contains a typed value. This class needs to be mapped into a concrete platform specific type.

2.10 Exceptions

All exceptions are placed in the Exception subpackage of the Deployment and Configuration package.

2.10.1 PackageError

Description



The <ClassName>PackageError exception is raised by the **installPackage** operation of the <ClassName>RepositoryManager if an internal error is detected in the package. (Potential reasons include the non-existence of a referenced file, or unresolved subcomponent references in an assembly.)

Attributes

label: **Sequence(String)**	Identifies a location in the package where the error occured.
reason: **String**	A human-readable description of the problem.

Associations

No associations.

Constraints

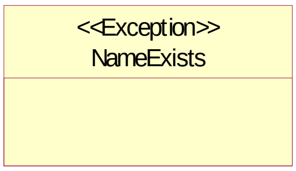
No constraints.

Semantics

The **label** attribute contains the **label** attributes from the elements in the hierarchy defined by the <ClassName>PackageConfiguration, **ComponentPackageDescription**, **ComponentImplementationDescription**, **SubcomponentInstantiationDescription**, <ClassName>AssemblyConnectionDescription, <ClassName>AssemblyPropertyMapping and <ClassName>ImplementationArtifactDescription elements to locate the problem as precisely as possible.

2.10.2 NameExists

Description



The <ClassName>NameExists exception is raised by the **installPackage** and **createConfiguration** operations of the <ClassName>RepositoryManager if a <ClassName>PackageConfiguration with the to-be-created name already exists in the repository.

Attributes

No attributes.

Associations

No associations.

Constraints

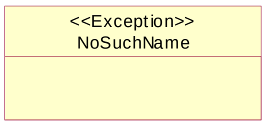
No constraints.

Semantics

No semantics.

2.10.3 NoSuchName

Description



The <ClassName>NoSuchName exception is raised by the **findConfigurationByLabel**, **createConfiguration**, **updateConfiguration** and **deleteConfiguration** operations of the <ClassName>RepositoryManager if there is no <ClassName>PackageConfiguration with the requested name in the repository.

Attributes

No attributes.

Associations

No associations.

Constraints

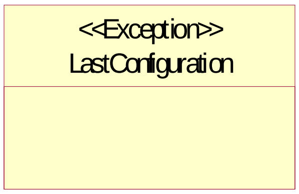
No constraints.

Semantics

No semantics.

2.10.4 LastConfiguration

Description



The <ClassName>LastConfiguration exception is raised by the **deleteConfiguration** operation of the <ClassName>RepositoryManager if the <ClassName>PackageConfiguration that is to be deleted is the last configuration for any package and the **deletePackage** parameter is false.

Attributes

No attributes.

Associations

No associations.

Constraints

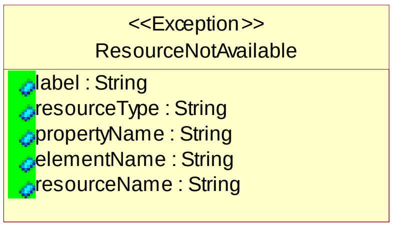
No constraints.

Semantics

No semantics.

2.10.5 ResourceNotAvailable

Description



The <ClassName>ResourceNotAvailable exception is raised by the **commitResources** operation of the <ClassName>TargetManager, by the **preparePlan** operation of the <ClassName>ExecutionManager or by the **startLaunch** operation of the <ClassName>ApplicationManager if a resource required by the plan is not available.

Attributes

label: String	Identifies the element in the plan whose resource requirement could not be satisfied.
resourceType: String	The type of resource that was requested using a <ClassName>Requirement element.
propertyName: String	The name of the property that could not be satisfied.
elementName: String	Identifies a <ClassName>Node, <ClassName>Interconnect or <ClassName>Bridge within the <ClassName>Domain.
resourceName: String	The name of a <ClassName>Resource or <ClassName>SharedResource within the <ClassName>Node, <ClassName>Interconnect or <ClassName>Bridge that was considered for matching the requirement.

Associations

No associations.

Constraints

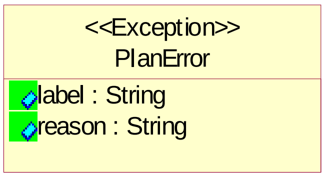
No constraints.

Semantics

The **label**, **resourceType** and **propertyName** uniquely identify a requirement in the plan. The **elementName**, **resourceName** and **propertyName** uniquely identify a requirement satisfier in the domain that failed to match the requirement. Note that **resourceName** can be the empty string if no <ClassName>RequirementSatisfier was found to match the **resourceType**.

2.10.6 PlanError

Description



The <ClassName>PlanError exception is raised by the `preparePlan` operation of the <ClassName>ExecutionManager if an inconsistency is detected in the plan. (E.g. an unresolved reference to a non-existent component instance.)

Attributes

label: **String**	Identifies an element of the <ClassName>DeploymentPlan where the error occured.
reason: **String**	A human-readable reason that describes the error.

Associations

No associations.

Constraints

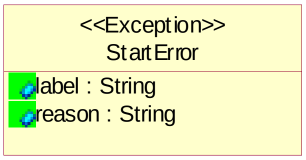
No constraints.

Semantics

This exception indicates that the plan is erroneous or inconsistent, i.e. the error is unrelated to the actual deployment.

2.10.7 StartError

Description



The <ClassName>StartError exception is raised if a problem occurred during deployment, either during preparation by the **preparePlan** operation of the <ClassName>ExecutionManager or during launch by the **startLaunch** operation of the <ClassName>ApplicationManager.

Attributes

label: **String**	Identifies an element of the <ClassName>DeploymentPlan where the error occured.
reason: **String**	A human-readable reason that describes the error.

Associations

No associations.

Constraints

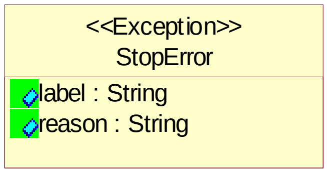
No constraints.

Semantics

Potential reasons include the inability to upload an artifact to a node or a failure during component instantiation.

2.10.8 StopError

Description



The <ClassName>StopError exception is raised if a problem occurred while terminating an application, either during the **terminate** operation of the <ClassName>ApplicationManager or during the **destroyManager** operation of the <ClassName>ExecutionManager.

Attributes

label: **String**	Identifies an element of the <ClassName>DeploymentPlan where the error occured.
reason: **String**	A human-readable reason that describes the error.

Associations

No associations.

Constraints

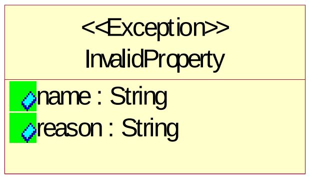
No constraints.

Semantics

This exception is raised if the problem is related to the “undeployment.” Potential reasons include the failure to stop a component instance.

2.10.9 InvalidProperty

Description



Attributes

name: **String**	The name of the property among the configProperty elements that caused the problem.
reason: **String**	A human-readable reason that describes the error.

Associations

No associations.

Constraints

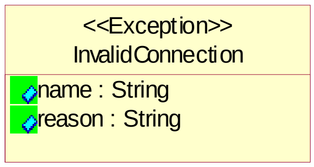
No constraints.

Semantics

The <ClassName>InvalidProperty exception is raised if an invalid property is passed to the **startLaunch** operation of the <ClassName>ApplicationManager. The problem can be that either the name does not match any of the component’s properties, or a type mismatch.

2.10.10 InvalidConnection

Description



Attributes

name: **String**	The name of the property among the configProperty elements that caused the problem.
reason: **String**	A human-readable reason that describes the error.

Associations

No associations.

Constraints

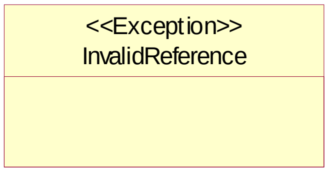
No constraints.

Semantics

The <ClassName>InvalidConnection exception is raised if an invalid connection is passed to the **finishLaunch** operation of the <ClassName>Application. The problem can be that the name does not match any of the component's ports, a type mismatch, or a direction mismatch (i.e. an attempt to connect a provider port to another provider port).

2.10.11 InvalidReference

Description



Attributes

No attributes.

Associations

No associations.

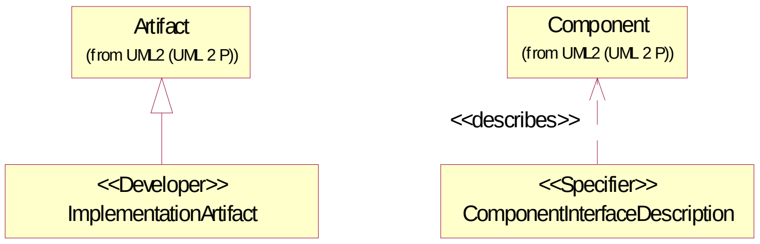
Constraints

No constraints.

Semantics

The <ClassName>InvalidReference exception is raised by the **destroyManager** operations of the <ClassName>ExecutionManager and <ClassName>NodeManager and the **destroyApplication** operation of the <ClassName>ApplicationManager if the <ClassName>ApplicationManager or <ClassName>Application reference is not known in this context. This may be because the reference was created by a different context, or because of prior destruction.

2.11 Relations to Other Standards



The <ClassName>ImplementationArtifact is a specialization of the **Artifact** class in the UML 2 Partners submission to the UML 2 RFP. It adds a self-relationship to describe dependencies between **Artifact** instances.

The <ClassName>ComponentInterfaceDescription describes the features of a **Component** features that are relevant to the deployment process, such as property names and types and port names and types.

Both for **Artifact** and **Component**, the relation to the UML 2 Partners submission to the UML 2 RFP is weak; in both cases, it is through a dependency relationship (<ClassName>ImplementationArtifact is only referenced by a dependency with the «**describes**» stereotype from <ClassName>ImplementationArtifactDescription). **Artifact** and **Component** will therefore not show up in any code that is generated from the model.

Since UML 2 is not an adopted standard yet, and since neither **Artifact** nor **Component** exist in UML 1.4, the dependencies might need to be updated or removed in sync with future iterations of UML 2 submissions. Because of the weak dependencies, changes in UML 2 do not have any impact on the models this document.

Actor 3

Contents

This chapter includes the following topics.

The previous chapter defined the platform independent model for deployment and configuration. The data models are used by the management interfaces for data interchange, but all model elements are passive entities. Actors manipulate the data, are clients to the interfaces and enact the various phases of deployment. Usually, part of the actor will be implemented in software tools, aiding a (human) user in development and deployment of an application.

All actors defined by this specification are abstract. Some behavior is regulated, e.g. how data is to be processed by them, but the implementation of actors is left undefined. Some implementations of this specification might combine all actors into a single GUI, others could provide separate scripts. Some actors might be implicit parts of derived actors, others might be split across multiple sub-actors. While the deployment system described by the PIM requires actors acting as clients to perform the work of deployment and configuration, the descriptions in this section are not normative, but rather express the expected usage of the capabilities offered by the PIM. In particular, run time errors can be expected if this anticipated actor behavior is not followed. Since any bundling or communication or modularity between actors is completely undefined, constraints cannot be described that insist on the behavior described in this section.

There are three categories for actors, development, target and deployment, mirroring the model segmentation presented earlier. Actors in the first category are concerned with the various phases of implementing a component, starting with an interface design and eventually creating a component package. Actors in the deployment category take existing component packages, and deploy them into a target environment in order to create running applications. The only actor in the target category is the Domain Administrator.

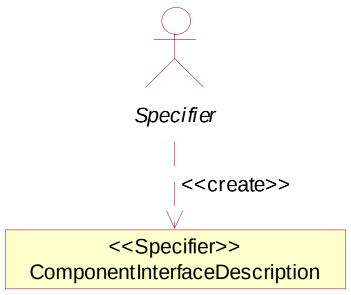
3.1 Development Actors Overview

The development of a component implementation involves the roles of Specifier, Developer, Assembler and Packager. The Specifier creates an interface specification. Developers create a monolithic implementation of that specification, or an Assembler creates an assembly based implementation from existing subcomponents. The Packager then wraps up one or more implementations of the component interface into a component package.

This process is circular, as component packages and/or interface specifications of subcomponents are inputs to the Assembler.

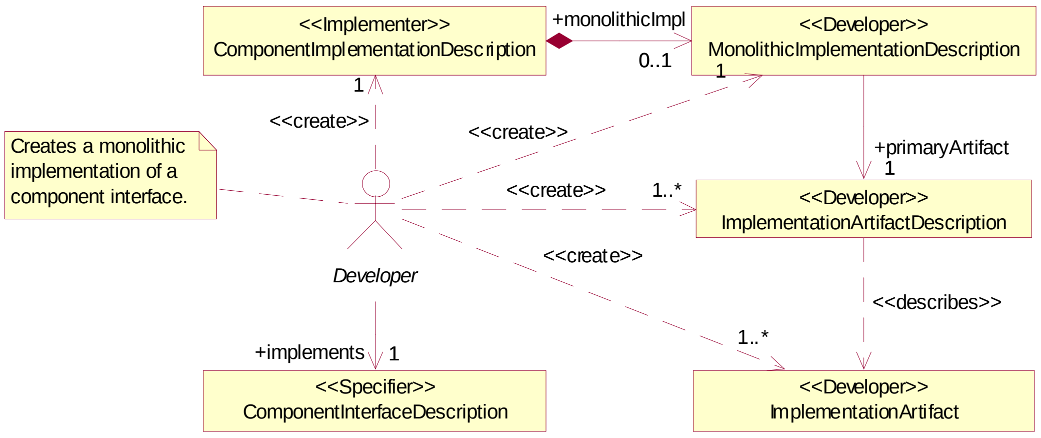
The above paragraph implies a bottom-up approach to component development, but that is not necessarily true, the flow of information can be reversed. An Implementer or Assembler can also work “downwards” from an existing component package in order to add new implementations to the package. An Assembler might then involve the Specifier in defining interface specifications for subcomponents.

3.2 Specifier



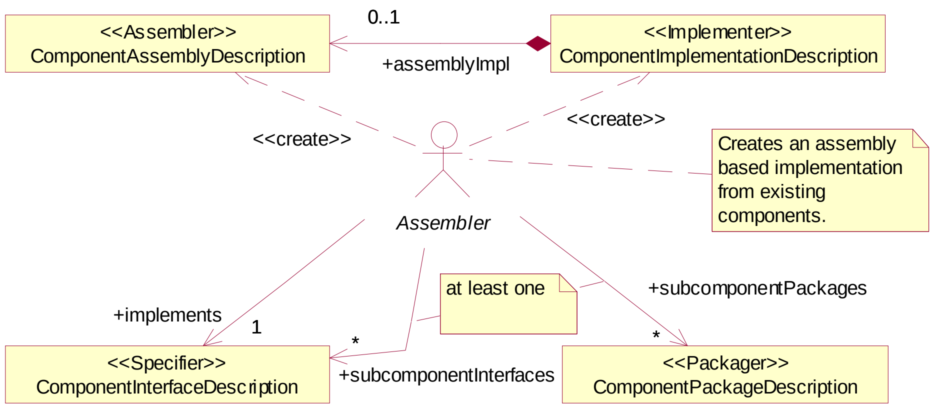
The Specifier creates an interface specification and generates a **ComponentInterfaceDescription** to describe the component interface, including its ports. Specifiers usually create other documents as well, such as PSM-specific interface descriptions (e.g. IDL files), behavioral models and system specifications, but the **ComponentInterfaceDescription** is the only piece that is captured in this model.

3.3 Developer



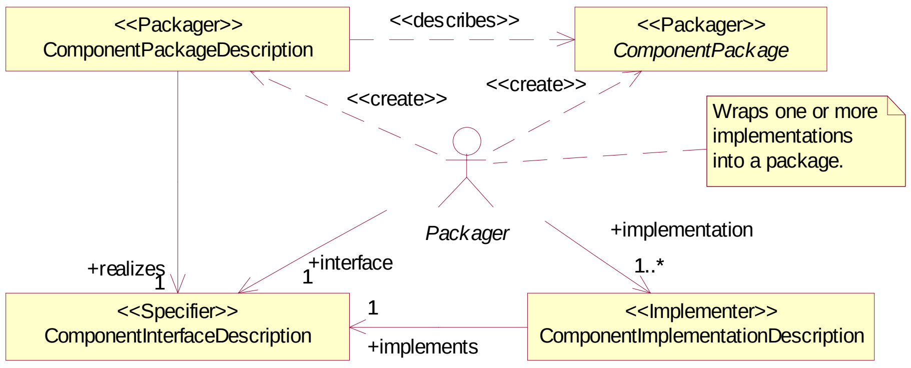
The Developer creates a monolithic implementation that satisfies a specific component interface. The Developer reads the Specifier's **ComponentInterfaceDescription** and creates an implementation contained in one or more implementation artifacts. For each **ImplementationArtifact**, the Developer then creates a matching **ImplementationArtifactDescription** that describes the artifact and its requirements on the target environment. The Developer then describes the component implementation as a whole by creating one **MonolithicImplementationDescription** and one **ComponentImplementationDescription** element.

3.4 Assembler


0..1
<<Assembler>>
ComponentAssemblyDescription
<<Implementer>>
ComponentImplementationDescription
+assemblyImpl
<<create>>
<<create>>
Creates an assembly based implementation from existing components.
Assembler
at least one
+subcomponentPackages
+implements
1
*
*
<<Specifier>>
ComponentInterfaceDescription
+subcomponentInterfaces
<<Packager>>
ComponentPackageDescription

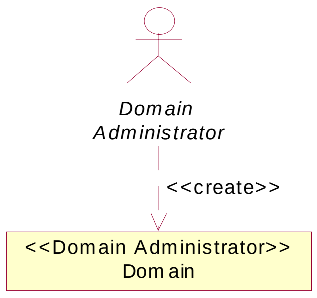
The Assembler creates an assembly based implementation of a specific component interface, using existing components as building blocks. The Assembler uses either interface descriptions for subcomponents from **ComponentInterfaceDescription** elements (expecting implementations for such interfaces to exist in the repository associated with the target domain) or concrete implementations for subcomponents from a **ComponentPackageDescription** (which implies an interface description). The Assembler configures subcomponents, interconnects them, and maps external ports and properties to ports and properties of subcomponents. The Assembler then creates a **ComponentAssemblyDescription** element to describe the assembly and a **ComponentImplementationDescription** to describe this component implementation.

3.5 Packager



The Packager wraps multiple implementations of the same component interface into a component package. The **ComponentInterfaceDescription** and one or more **ComponentImplementationDescription** elements are input to the packaging process. The Packager ensures that the implementations' component interfaces are compatible with the desired interface. The Packager then creates a **ComponentPackageDescription**, potentially assigning default values to properties. The Packager then creates a component package that wraps all relevant descriptors and implementation artifacts. This component package is then distributed to Repository Administrators.

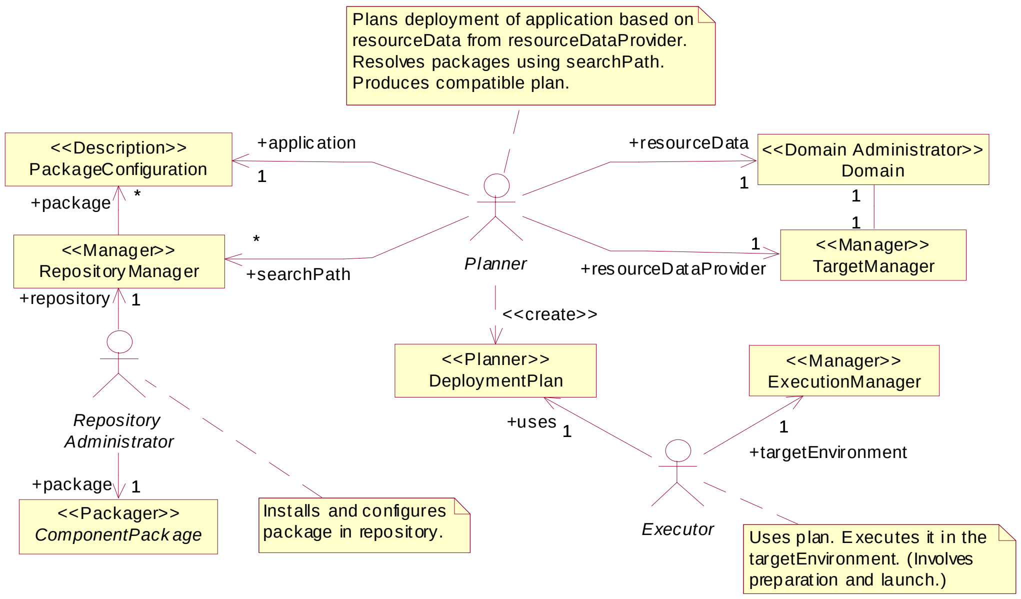
3.6 Domain Administrator



The Domain Administrator describes the local target environment and all its resources by creating a **Domain** element and then initializing a **TargetManager** with that information.

Note – In the future, the Domain Administrator role could be refined. Ideally, hardware providers would deliver descriptions for all pieces of a domain: nodes, interconnects, bridges, hardware devices etc. The Domain Administrator would then collect that information and create a specific domain configuration. For the moment, it is safe to assume that the job of describing a domain's resources ends up with the Domain Administrator.

3.7 Deployment Actors Overview


Plans deployment of application based on resourceData from resourceDataProvider. Resolves packages using searchPath. Produces compatible plan.
<<Description>> PackageConfiguration
+application
1
+resourceData
1
<<Domain Administrator>> Domain
1
1
+package
*
*
<<Manager>> RepositoryManager
+searchPath
Planner
1
+resourceDataProvider
<<Manager>> TargetManager
+repository
1
<<create>>
<<Planner>> DeploymentPlan
<<Manager>> ExecutionManager
Repository Administrator
+uses
1
1
+targetEnvironment
+package
1
<<Packager>> ComponentPackage
Installs and configures package in repository.
Executor
Uses plan. Executes it in the targetEnvironment. (Involves preparation and launch.)

The overview diagram above shows the three actors that are involved in the deployment of an application, the Repository Administrator, the Planner and the Executor. The Repository Administrator receives component packages from the Packager and installs them in the local repository using the **RepositoryManager** interface. The Planner matches an implementation's requirements against available resources and creates a specific **DeploymentPlan**. The Executor uses the **DeploymentPlan** and contacts the **ExecutionManager** in order to execute the deployment and to instantiate the application. More detail is provided in the upcoming sections.

3.8 Repository Administrator

The Repository Administrator installs a component package into a repository, and then configures the component packages within the repository.

The Repository Administrator has access to a component package via URL, and to a **RepositoryManager** via reference. The Repository Administrator calls the **installPackage** operation of the **RepositoryManager**, passing the URL of the component package. A user may provide a label for the new **PackageConfiguration**.

After installing a package in the repository, the configuration for that package may optionally be updated, or new configurations can be created. In order to update or create a configuration, the user provides configuration and selection properties, and the Repository Administrator can then use the **createConfiguration** or **updateConfiguration** operation of the **RepositoryManager** to effect the update or creation of a **PackageConfiguration**.

3.9 Planner

The Planner supports planning the deployment of an application.

The Planner has access to a specific **PackageConfiguration** via a repository reference and a name: the Planner uses the **findConfigurationByName** operation of the **RepositoryManager** to retrieve the description of the application that is to be deployed. A user might provide zero or more references to **RepositoryManager** instances as a search path to resolve **ComponentPackageReference** references in the component package. To resolve such a reference, the Planner passes the **specificType** from the **ComponentPackageReference** to the **findLabelsByUID** operation of each **RepositoryManager** in the search path and selects an appropriate configuration among all available configurations using implementation defined means. The Planner then retrieves resource data from a **TargetManager** using either the **getAllResources** or **getAvailableResources** operation. From this information, the Planner produces a **DeploymentPlan** that details a valid deployment of the application into the domain.

The Planner selects a valid **DeploymentPlan** using implementation defined means. Usually, there will be many possibilities to deploy an application into a domain, some of them equivalent – e.g. permutations of distributing component instances among homogeneous nodes, – some of them can be considered better than others – e.g. distributing computation-intensive component instances across multiple nodes rather than executing them on a single node. Selecting plans that are more appropriate than others in a given context is a quality of implementation issue, possibly influenced by user input and feedback.

A valid **DeploymentPlan** describes a deployment of an application using concrete implementations that match requested selection properties, and an assignment of these implementations to nodes so that node and interconnection resources match or exceed the requirements of component and connection instances that are deployed on them.

3.9.1 Finding Valid Deployments

To find a valid deployment, the Planner may have to consider all potential decompositions of an application, and all potential distributions. One possible algorithm is to consider a decision tree where inner nodes mark selections of specific implementations within a component package. The leaves of the tree then represent decompositions of the application into monolithic implementations. For each decomposition, the Planner then has to consider all possibilities for distributing component instances among all nodes until a valid deployment is found. Pseudo code for this algorithm follows.

1. Initialize a “decision queue” with the top-level package that is to be deployed. This queue will contain packages for which we still have to decide on an implementation. Recurse into the algorithm, initializing it with the one-element decision queue, starting at step 2. If the recursion fails, there is no valid deployment.

2. Remove the first element from the queue, which identifies a **ComponentPackageDescription**.

3. For each concrete implementation in the package, go to step 4 to find a valid deployment. If that fails, backtrack.

4. Match the capabilities of this **ComponentImplementationDescription** against the relevant selection requirements (see below). On the top level, i.e. for the implementations of the top-level component, selection requirements are found in the **PackageConfiguration**. On other levels, i.e. for implementations of subcomponents in an assembly, the selection requirements are found in the **SubcomponentInstantiationDescription**. If they are not compatible, return to step 3 and continue iterating over other implementations in this package.

5. If the implementation is assembly-based, then add the packages that provide implementations for its subcomponents to the decision queue.

6. If the decision queue is not empty, then the application is not fully decomposed yet. Recurse to step 2. If recursion fails, return to step 3.

7. If the decision queue is empty, then the application has been fully decomposed into monolithic implementations by the decisions made in step 3. The Planner now has to consider potential instantiations.

8. Iterate over all permutations of assigning component instances to nodes. For each permutation, go to step 9 to see whether it identifies a valid deployment. If that fails, backtrack.

9. For each component instance, consider the node it has been assigned to. Match the requirements defined by its monolithic implementation against the node's resources (see below). If that fails, return to step 8 to consider other permutations.

10. For each connection between component instances, match its connection requirements against the interconnect and bridge resources that provide the connection between the nodes that the component instances have been assigned to (see below). If there is no path between the nodes, or if the interconnects and bridges are not capable of hosting the connection, return to step 8.

11. Otherwise, the deployment is valid.

This specification does not impose any requirements on the Planner implementation. The algorithm above is designed to find a valid deployment if one exists. It has been included for informative purposes and is not normative. Obviously, there are many techniques for narrowing the search space and for considering more likely implementations and permutations first, but still, the number of possibilities might be too large to be practical. Planners are not required to traverse the full search space – that's a quality of implementation issue. Planners are also free to either stop after finding a first valid deployment or to continue searching and to select among valid deployments – possibly with user feedback.

Steps 4, 9 and 10, the matching of selection properties and the matching of requirements against resources, are defined in the following sections.

Note – Steps 2, 3 and 5 assume that in order to find a concrete implementation for a component, only a single package is considered. However, Planner implementations might consider multiple packages when resolving **ComponentPackageReference** elements. Again, this is implementation specific.

3.9.2 Matching Selection Requirements

Both **PackageConfiguration** and **SubcomponentInstantiationDescription** define selection requirements that are matched against implementation capabilities in the **ComponentImplementationDescription** for all implementations in the referenced **ComponentPackageDescription**.

For each **Requirement**, the Planner checks whether the **ComponentImplementationDescription** has a **Capability** whose **resourceType** attribute includes the **resourceType** attribute of the **Requirement**. If not, then the implementation cannot satisfy the requirements.

The **Requirement** is then matched against the **Resource** as described below.

3.9.3 Matching Implementation Requirements

A component instance's requirements are defined as the sum of all deployment requirements in its **MonolithicImplementationDescription**, the **ImplementationArtifactDescription** of its primary artifacts and all directly or indirectly dependent **ImplementationArtifactDescription** elements (excluding duplicates). The "sum" of all requirements is the concatenation of all **Requirement** elements into a single list.

For each **Requirement**, the Planner checks whether the **Node** has a **Resource** (or **SharedResource** – resources and shared resources are treated the same) whose `resourceType` attribute includes the **resourceType** attribute of the **Requirement**. If not, then the **Node** is not capable of hosting the component implementation.

The **Requirement** is then matched against the **Resource** as described below.

3.9.4 Matching Connection Requirements

Connection requirements are described as part of an assembly in the **deployRequirement** attribute of the **AssemblyConnectionDescription**. Connections between two component ports can be made up of multiple segments if the two components belong to different assemblies, e.g. two segments to connect the components to external ports of their respective assemblies, and another segment to connect the two components (that are implemented by the assemblies) in the assembly-based implementation of a supercomponent. In that case, the requirements for the connection is the sum of all deployment properties of all its segments. The "sum" of all requirements is the concatenation of all **Requirement** elements into a single list.

Note – Considering point-to-point connections between two ports is the worst-case scenario. In some domains, if a connection has more than two endpoints, part or all of the communication path could be shared – e.g. if events are broadcast using UDP. Planners that are aware of this situation can account for capacities appropriately.

Connection requirements must be matched against the resources of the interconnects and bridges that the connection is routed over, as defined by the communication path between the nodes that the components that are the endpoints to the connection are instantiated on.

Note – This specification assumes that a single communication path is implied by its two endpoints.

For each **Requirement**, the Planner checks whether all **Interconnect** and **Bridge** elements in the communication path have a **Resource** whose **resourceType** attribute includes the **resourceType** attribute of the **Requirement**. If not, then routing the connection is not possible.

The **Requirement** is then matched against all these **Resource** elements as described below. If any match fails, then routing the connection is not possible.

3.9.5 Matching a Resource against a Requirement

For every **Property** that is part of the **Requirement**, there must be a **SatisfierProperty** among the property elements of the **Resource** whose **name** attribute equals the **name** attribute of the requirement's property. If there is no **SatisfierProperty** of matching name, then the **Resource** cannot satisfy the **Requirement**.

Each **Property** is then matched against the **SatisfierProperty** according to the rules set forth for the kind of **SatisfierProperty**, as described in the documentation for **SatisfierPropertyKind**, to determine if the **Resource** meets this specific requirement.

The **Resource** meets the **Requirement** if and only if the above test succeeds for all **Property** elements that are part of the **Requirement**.

3.10 Executor

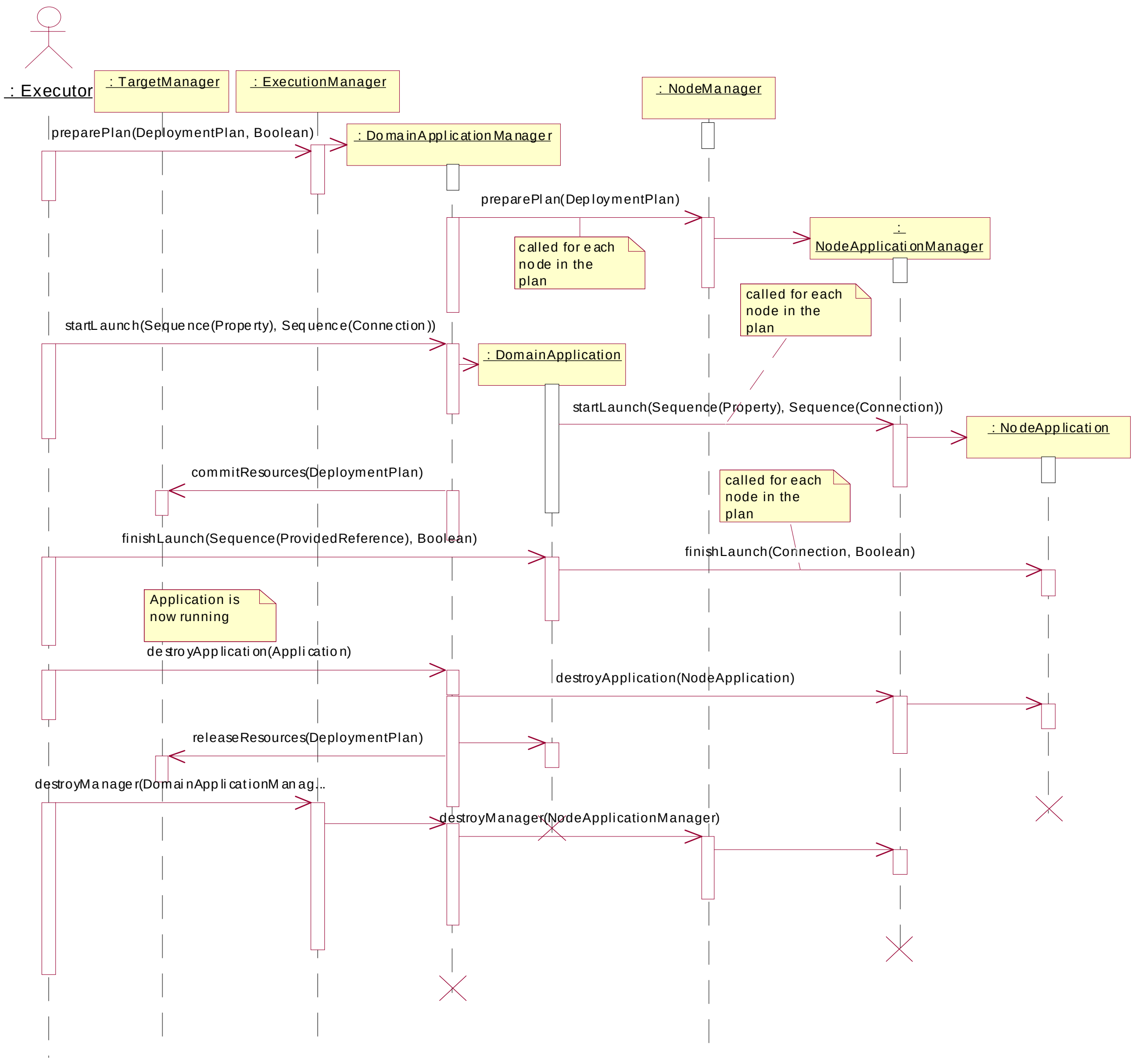
The Executor supports preparation of a **DeploymentPlan** and the launch of the application, possibly, but not necessarily, in a single step.

For preparation, the Executor reads the **DeploymentPlan** and passes it to the `preparePlan` operation of the **ExecutionManager**. The Executor stores the **DomainApplicationManager** reference that is returned.

To launch an application, the Executor remembers the **DomainApplicationManager** reference that was the result of preparation, and calls the `startLaunch` operation, passing configuration properties if desired. The **DomainApplicationManager** returns a **DomainApplication** reference and the connections that are provided by the application on external ports.

The Executor then calls the **finishLaunch** operation on the **DomainApplication**, passing connections to the application's external ports if desired.

The Executor can either set the start parameter to the **finishLaunch** operation to true in order to start the **DomainApplication**, or it can later call the **start** operation separately.


: Executor
: TargetManager
: ExecutionManager
: NodeManager
: DomainApplicationManager
preparePlan(DeploymentPlan, Boolean)
preparePlan(DeploymentPlan)
: NodeApplicationManager
called for each node in the plan
called for each node in the plan
startLaunch(Sequence(Property), Sequence(Connection))
: DomainApplication
startLaunch(Sequence(Property), Sequence(Connection))
: NodeApplication
commitResources(DeploymentPlan)
called for each node in the plan
finishLaunch(Sequence(ProvidedReference), Boolean)
finishLaunch(Connection, Boolean)
Application is now running
destroyApplication(Application)
destroyApplication(NodeApplication)
releaseResources(DeploymentPlan)
destroyManager(DomainApplicationManag...
destroyManager(NodeApplicationManager)

The above figure shows the sequence of events that are exchanged between the Executor and the deployment system as well as events within the domain.

UML Profile for D+C Tool Support 4

Contents

This chapter includes the following topics.

"Proposals may define textual or graphical notation(s) for the description of software and hardware infrastructures of distributed execution environments as well as to express configuration and deployment constraints for components or assemblies of components. If a proposal does so, it must define the relationship between the models provided by it and the notation(s) defined."

The main objectives of the UML Profile for D&C Tool Support are:

- to define the notation necessary to support the component-based application development process and target environment description, as described in chapter 2
- to enable the automatic generation of D&C descriptors from component assembly and target models.

The UML Profile for D&C Tool Support provides tool vendors with the foundation they need to develop UML tools that support the deployment and configuration of component-based distributed applications. The current D&C specification is composed of three main parts: Component, Target, and Deployment. This profile addresses the first two. The description of the deployment infrastructure is outside the scope of the current UML Profile for D&C Tool Support, and will need to be addressed seperately. Currently UML allows deployment planners to define an explicit deployment plan by statically associating component with nodes.

The concepts and notation defined by this profile allows application developers to use UML to completely model the configuration of components, the assembly of components from other components, and the target environments into which components can be deployed.

The development of tools to support the D&C specification, based on the UML Profile for D&C Tool Support, offers many important advantages:

- enables the integration of model validation techniques that will allow eliminating errors at the application design stage
- eliminates errors in the production of descriptors
- makes the component and target models independent of the specific format of the descriptors, which allow changing the format without having to change the models
- enables the integation with existing UML-based MDA tools

4.1 Structure of the Profile

The UML Profile for D&C Tool Support is defined using the profiles mechanism defined in UML 2.0.

The UML Profile for D&C Tool Support is composed of a set of stereotypes that are defined as extensions of UML 2.0 metaclasses. In particular, the D&C Profile for Tool Support extends metaclasses defined in the UML 2.0 Component, Composite Structures, and Deployment packages. The dependencies between the D&C Profile for Tool Support and UML 2.0 packages is illustrated in Figure 4-1.

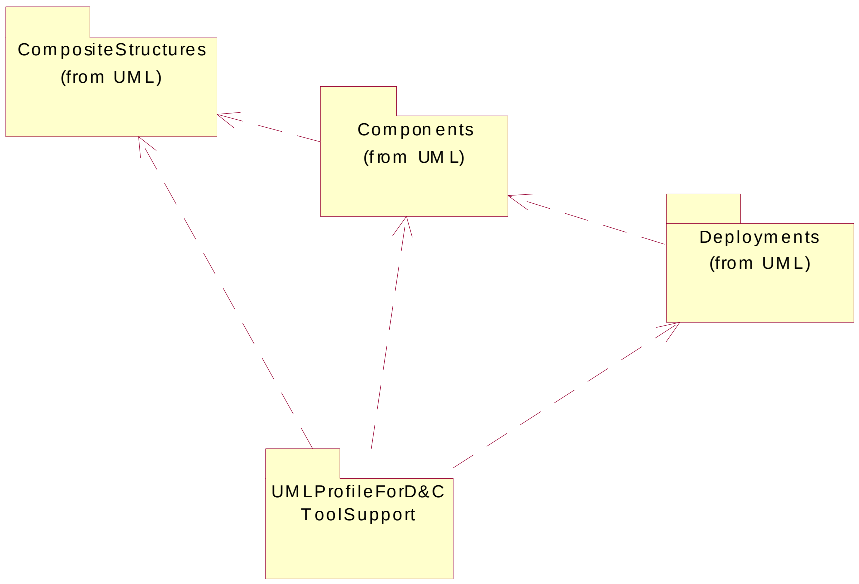


Figure 4-1 Dependencies between the UML Profile for D&C Tool Support and UML 2.0 packages

The set of stereotypes that compose the UML Profile for D&C Tool Support are grouped in two disctinct packages: Component and Target. The Component package defines the set of stereotypes that are used to model a component-based distributed application. The Target package defines the set of stereotypes that are used to model a distributed deployment target.

The content of these packages is defined in the next two sections (Section 4.2, “Package Components,” on page 4-4 and Section 4.3, “Package Targets,” on page 4-14). The dependencies between the Component and Target packages and the UML 2.0 packages are illustrated in Figure 4-2.

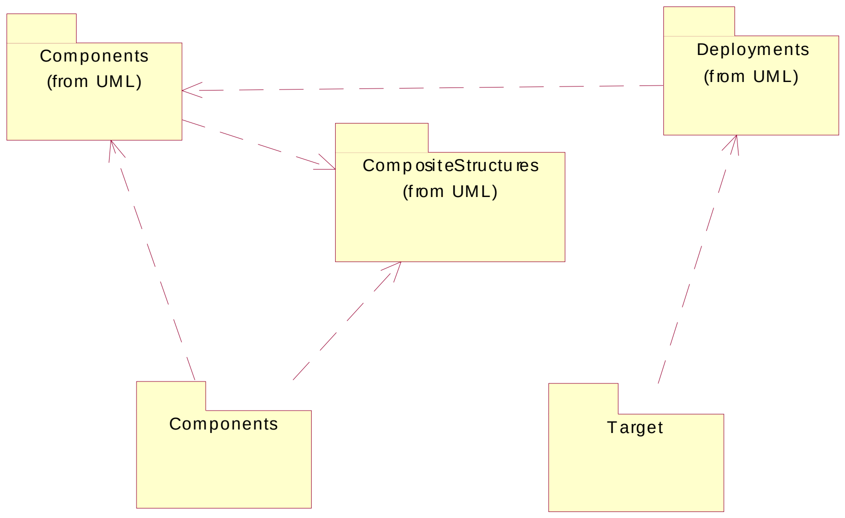
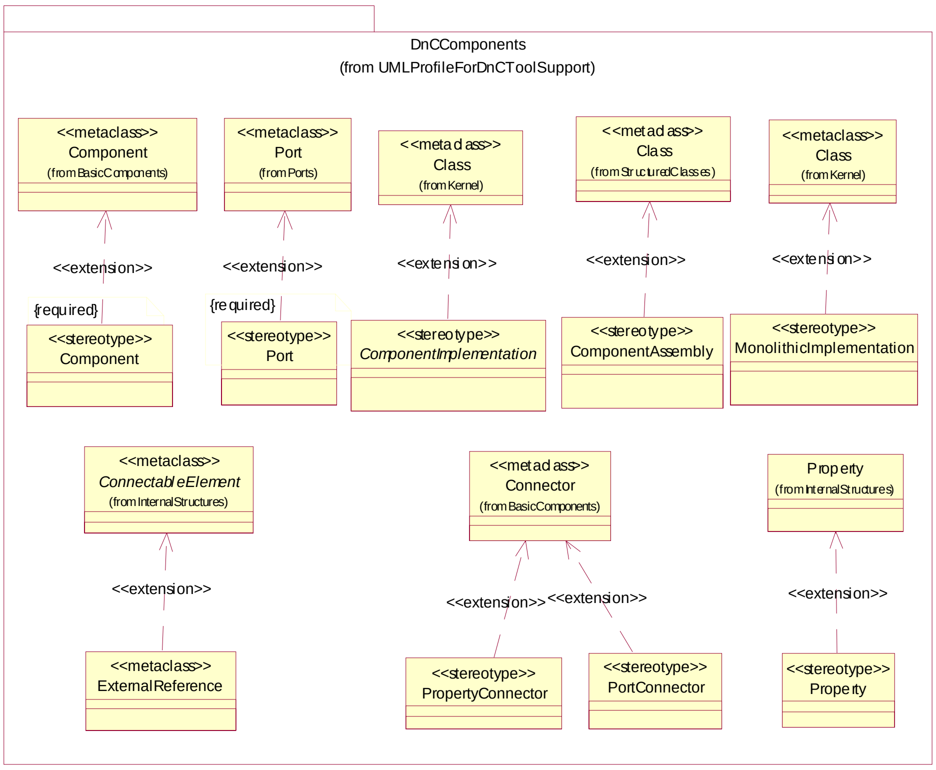


Figure 4-2 Dependencies between the Component and Target packages and UML 2.0 packages

4.2 *Package Components*

The Component package defines the set of stereotypes that are used to model a component-based distributed application. The list of stereotypes currently defined in the Component package includes: Component, ComponentAssembly, Connection, Port, and Artifact.

This section defines the set of stereotypes contained in the package Components.



DnCComponents
(from UMLProfileForDnCToolSupport)
<<metaclass>> Component (from BasicComponents)
<<metaclass>> Port (from Ports)
<<metaclass>> Class (from Kernel)
<<metaclass>> Class (from StructuredClasses)
<<metaclass>> Class (from Kernel)
<<extension>>
{required}
<<stereotype>> Component
<<stereotype>> Port
<<stereotype>> ComponentImplementation
<<stereotype>> ComponentAssembly
<<stereotype>> MonolithicImplementation
<<metaclass>> ConnectableElement (from InternalStructures)
<<metaclass>> Connector (from BasicComponents)
Property (from InternalStructures)
<<metaclass>> ExternalReference
<<stereotype>> PropertyConnector
<<stereotype>> PortConnector
<<stereotype>> Property

Figure 4-3 Components package

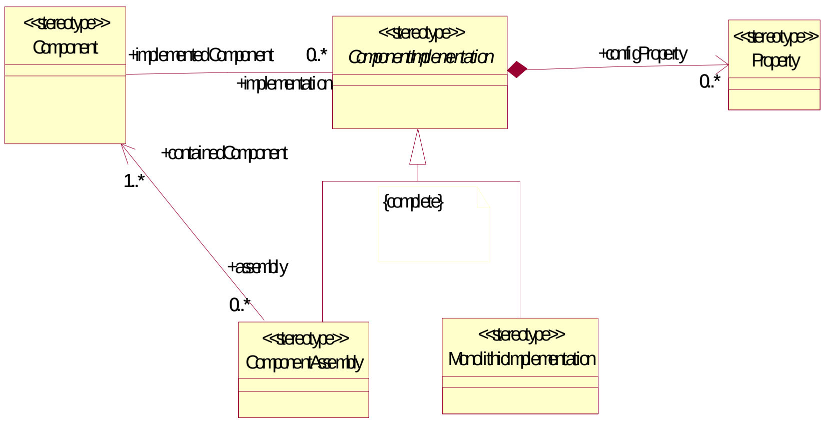


Figure 4-4 Component implementation relationships

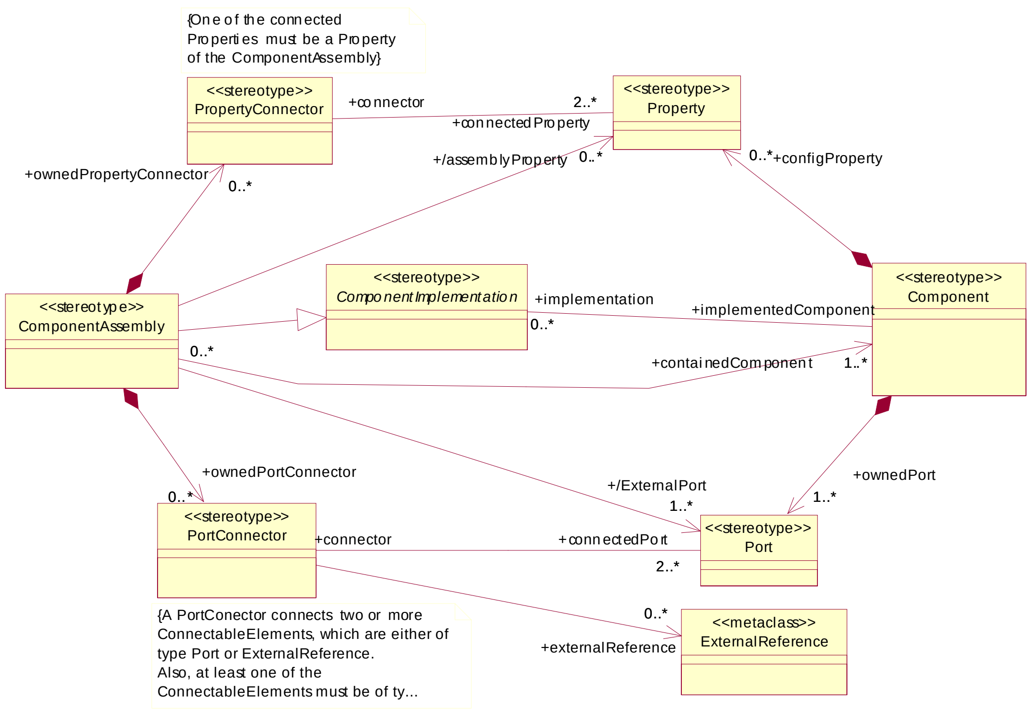

{One of the connected Properties must be a Property of the ComponentAssembly}
<<stereotype>> PropertyConnector
<<stereotype>> Property
+connector
2..*
+connectedProperty
+/assemblyProperty
0..*
0..*+configProperty
+ownedPropertyConnector
0..*
<<stereotype>> ComponentAssembly
<<stereotype>> ComponentImplementation
+implementation
0..*
+implementedComponent
<<stereotype>> Component
0..*
+containedComponent
1..*
+ownedPortConnector
0..*
+/ExternalPort
1..*
+ownedPort
1..*
<<stereotype>> PortConnector
+connector
+connectedPort
2..*
<<stereotype>> Port
{A PortConector connects two or more ConnectableElements, which are either of type Port or ExternalReference. Also, at least one of the ConnectableElements must be of ty...
0..*
+externalReference
<<metaclass>> ExternalReference

Figure 4-5 ComponentAssembly Stereotype

4.2.1 *Capability*

Description

Capability is used to describe an implementation’s capabilities, which are matched against selection requirements.

Attributes

name: String	An optional name for the requirement satisfier.
resourceType: Sequence(String)	The resource types that can be satisfied by this satisfier.

Associations

No associations

4.2.2 Component (Stereotype)

Description

The Component metaclass extends the UML Component metaclass (from UML2.0::Components). In UML 2.0, a component is defined in terms of its set of ports and has references to its realizations.

The Component stereotype is defined as “required”, which means that every instance of the Component metaclass must be associated with an instance of the Component stereotype.

Attributes

label: String	An optional human-readable label for the component.
UUID: String	An optional unique identifier for this component.

Associations

implementation: ComponentImplementation [0..*]	References the Classifiers of which the Component is an abstraction, i.e. that realize its behavior. This association renames the “realization” association owned by Component (from UML2.0::Components::Component).
configProperty: Property [*]	Contains the set of configurable properties of the component. These configuration properties are used to configure the component once instantiated. This allows the definition of configuration properties in a package regardless of which implementation is chosen. configProperty is a subset of the ownedAttribute association of Component (inherithed from UML2.0::CompositeStructures::InternalStructures::StructuredClassifier).
ownedPort: Port [*]	Contains the set of ports of the component.These configuration properties are used to configure the component once instantiated. This allows the definition of configuration properties in a package regardless of which implementation is chosen. ownedPort is a renaming of the ownedPort association of Component (inherithed from UML2.0::CompositeStructures::Ports::EncapsulatedClassifier).

Note – Definition. Component (from UML2.0::Components): A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type, whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics). One component may therefore be substituted by another only if the two are type conformant. Larger pieces of a system's functionality may be assembled by reusing components as parts in an encompassing component or assembly of components, and wiring together their required and provided interfaces. A component is modeled throughout the development life cycle and successively refined into deployment and run-time. A component may be manifest by one or more artifacts, and in turn, that artifact may be deployed to its execution environment. A deployment specification may define values that parameterize the component's execution.

4.2.3 ComponentAssembly (Stereotype)

Description

In spite of the fact that UML 2.0 allows for the recursive definition of components in terms of subcomponents (based on the fact that a UML 2.0 Component is a specialization of UML2.0::StructuredClass::Class), the concept of component assembly is not explicitely defined in UML 2.0. The ComponentAssembly stereotype specilizes the UML 2.0 Class metaclass from StructuredClasses (UML2.0::CompositeStructures::StructuredClasses). It is a subclass of the ComponentImplementation stereotype.

A ComponentAssembly is a classifier whose behavior is fully described by the collaboration of a set of components. A ComponentAssembly is defined in terms of a set of components (subcomponents) and the set of connections that connect components.

A ComponentAssembly is defined as an implementation of a Component.

A ComponentAssembly also has a two derived attributes: ports, that contains the set of external ports of the assembly implements, and properties, that contains the set of properties of the assembly. These two attributes are derived from the component the assembly implements. The ports and properties of the implemented component must be allocated to ports and properties of sub-components contained in the ComponentAssembly.

Attributes

No additional attributes.

Associations

/assemblyProperty: Property [0..*]	Contains the set of properties of the assembly. This association is derived from the Component the assembly implements.
/externalPort: Port [0..*]	Contains the set of external ports of the assembly. This association is derived from the Component the assembly implements.
containedComponent: **Component** [1..*]	Describes the set of Components contained in the ComponentAssembly (i.e. subcomponents). This association is a subset of the "role" association owned by the StructuredClassifier (UML2.0::CompositeStructures::InternalStructures::Structured-Classifier).
ownedPortConnector: **PortConnector** [*]	Describes the set of PortConnectors owned by the ComponentAssembly. This association is a subset of the ownedConnector as-

sociation owned by UML2.0::CompositeStructures::InternalStructures::StructuredClassifier

ownedPropertyConnector: **PropertyConnector** [*]
Maps the external properties of the component that is implemented by the assembly to properties of subcomponent instances. Describes the set of PropertyConnectors owned by the ComponentAssembly. This association is a subset of the ownedConnector association owned by UML2.0::CompositeStructures::InternalStructures::StructuredClassifier

4.2.4 ComponentImplementation (Stereotype)

Description

The ComponentImplementation stereotype is an extension of the UML 2.0 Class metaclass (from UML2.0::Kernel). A ComponentImplementation is an abstract class that contains the defines the attributes and associations that are common to the different types of component implementations (MonolithicImplementation and ComponentAssembly).

A ComponentImplementation describes a specific implementation of a component interface. This implementation can be either assembly based or monolithic. The ComponentImplementation may contain configuration properties that are used to configure each component instance ("default values"). Implementations may be tagged with user-defined capabilities. Administrators can then select among implementations using selection requirements; Assemblers can place requirements on implementations.

Attributes

capacityt: Sequence(Capacity)	Tags that can be used to discriminate between implementations.

Associations

deployRequirement: Requirement [1..*]
Requirements that are matched against node resources at deployment time.

4.2.5 ExternalReference (Stereotype)

Description

The ExternalReference stereotype is an extension of the UML 2.0 ConnectableElement metaclass (from UML2.0::CompositeStructures::InternalStructures). It identifies a location outside the assembly as an endpoint of a PortConnector. Whether the endpoint is a provider or user port is implied by the URL, and its type is assumed to be compatible with the connection.

Attributes

location: **URL**	References a port outside of the assembly that is to be an endpoint of this connection, which is resolved at execution time.

Associations

No associations.

4.2.6 PortConnector (Stereotype)

Description

The PortConnector stereotype is an extension of the UML 2.0 Connector metaclass (from UML2.0::Components::BasicComponents). A PortConnector connects a set of compatible ports.

Attributes

label: String	Optionally identifies this connection within its assembly. May be used or generated by visual design tools.

Associations

connectedPort: **Port** [1..*]	The set of Ports connected by the PortConnector. This association is a subset of the "end" association owned by UML2.0::Composite-Structures::InternalStructures::Connector.
externalReference: **externalReference** [*]	The set of ExternalReferences connected by the PortConnector. This association is a subset of the "end" association owned by UML2.0::CompositeStructures::InternalStructures::Connector.

4.2.7 Constraints

A PortConector connects two or more ConnectableElements, which are either of type Port or ExternalReference.

Also, at least one of the ConnectableElements must be of type Port.

Note – Definition. Connector (from UML2.0::Components::BasicComponents): The connector concept is extended in the Components package to include interface based constraints and notation. A delegation connector is a connector that links the external contract of a component (as specified by its ports) to the internal realization of that behavior by the component's parts. It represents the forwarding of signals (operation requests and events) : a signal that arrives at a port that has a delegation connector to a part or to another port will be passed on to that target for handling. An assembly connector is a connector between two components that defines that one component provides the services that another component requires. An assembly connector is a connector that is defined from a required interface or port to a provided interface or port.

Note – One of the issues in the D&C is that a single connector can at the same time connect ports of peer components in an assembly and ports of internal components to external ports, i.e. delegation ports. So according to the UML 2.0 spec, we have connectors that have both a delegation connector capability and an assembly connector capability. The D&C concept of PortConnector is based on the ECAD (circuit design, netlist) model. It fully expresses the idea that a set of ports can be connected together just like a "signal" (say "the reset signal") can be connected to many "pins" of the components (chips) of a circuit. This allows the expression of connections that are point to point (one provider and one user) as well as those with multiple users (line many clients for one server, many event producers for one consumer), multiple providers (like a multicast channel), or multiple of both (like a multicast event channel with multiple listeners). Also, in network systems, you want to talk about a flow that represents the traffic between a set of users and providers so you can plan, manager, and configure it as a whole. If the only means of expression is point to point connections, there is no way to talk about the aggregate "connection". This "richness" has been used in network, circuit, and chip design systems for decades.

4.2.8 PropertyConnector (Stereotype)

Description

The PropertyConnector stereotype is an extension of the UML 2.0 Connector metaclass (from UML2.0::Components::BasicComponents). A PropertyConnector connects properties of a ComponentAssembly to properties of sub-Components.

Attributes

label: String	Optionally identifies this connection within its assembly. May be used or generated by visual design tools.

Associations

connectedProperty: **Property** [2..*]	The set of Properties connected by the PropertyConnector. This association is a subset of the “end” association owned by UML2.0::CompositeStructures::InternalStructures::Connector.

Constraints

One of the connected Properties must be a Property of the ComponentAssembly.

4.2.9 MonolithicImplementation (Stereotype)

Description

The MonolothicImplementation stereotype is an extension of the UML 2.0 Class metaclass (from UML2.0::Kernel). It is a subclass of the ComponentImplementation stereotype. A MonolithicImplementation is a class that contains the implementation of a component.

Attributes

deployRequirement: Requirement [1..*]	Requirements that are matched against node resources at deployment time.

Associations

No additional associations.

4.2.10 Port (Stereotype)

Description

The Port stereotype is an extension of the UML 2.0 Port metaclass (from UML2.0::CompositeStructure::Ports).

The Port stereotype is defined as “required”, which means that every instance of the Port metaclass must be associated with an instance of the Port stereotype.

Attributes

name: **String**	The name of the port.
UID: **String**	The primary type of the port.
supportedType: **Sequence(String)**	All types supported by this port, including the primary and inherited types. All of the types listed in this attribute are acceptable for a connection.
provider: **Boolean**	Identifies whether the port acts in the role of provider or user, for any connection attached to it.
exclusiveProvider: **Boolean**	If set to true, then this port expects that there is at most one provider on the connection that it is an endpoint to.
exclusiveUser: **Boolean**	If set to true, then this port expects that there is at most one user on the connection that it is an endpoint to.
optional: **Boolean**	Identifies whether connecting this port is optional or mandatory.

Associations

No additional associations.

Note – Restriction. In UML 2.0, a Port can be associated with both required and provided interfaces. In this D&C specification, a Port is restricted to be associated with either required interfaces (user Port) or provided interfaces (provider Port). An OCL constraint could be added to formally express this restriction.

4.2.11 Property (Stereotype)

Description

The Property stereotype is an extension of the UML 2.0 Property metaclass (from UML2.0::CompositeStructures::InternalStructures). A Property has a name and a typed value. It is used to carry named and typed values in various places. In the context of D&C, components have configuration properties.

Attributes

No additional attributes.

Associations

No additional associations.

4.2.12 Requirement

Description

Requirements are used to express the fact that an implementation artifact or connection has requirements that must be fulfilled by resources in the target environment. The resource type must match the type of a resource.

Attributes

resourceType: **String**	Identifies the resource type.

Associations

properties: **Property** [*]	Properties associated with the resource.

4.3 Package Targets

The Target package defines the set of stereotypes that are used to model a distributed deployment target. The list of stereotypes currently defined includes: Bridge, CommunicationPath, Domain, Interconnect, Node, Resource, and SharedResource.

This section defines the set of stereotypes contained in the package Targets.

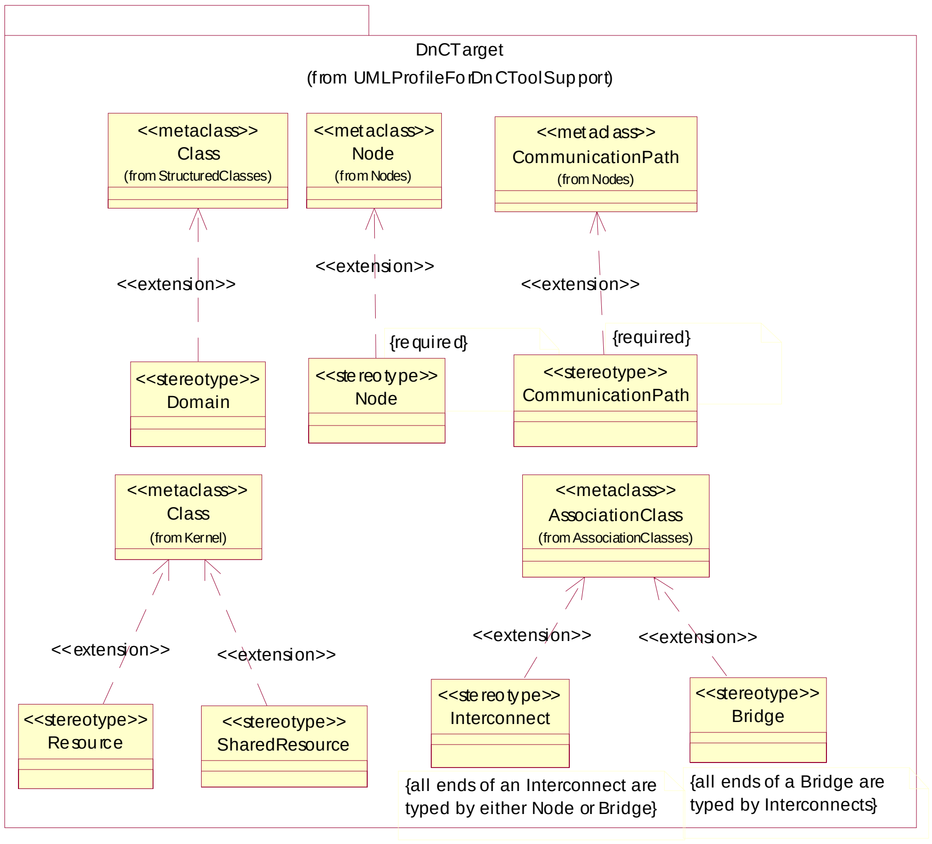
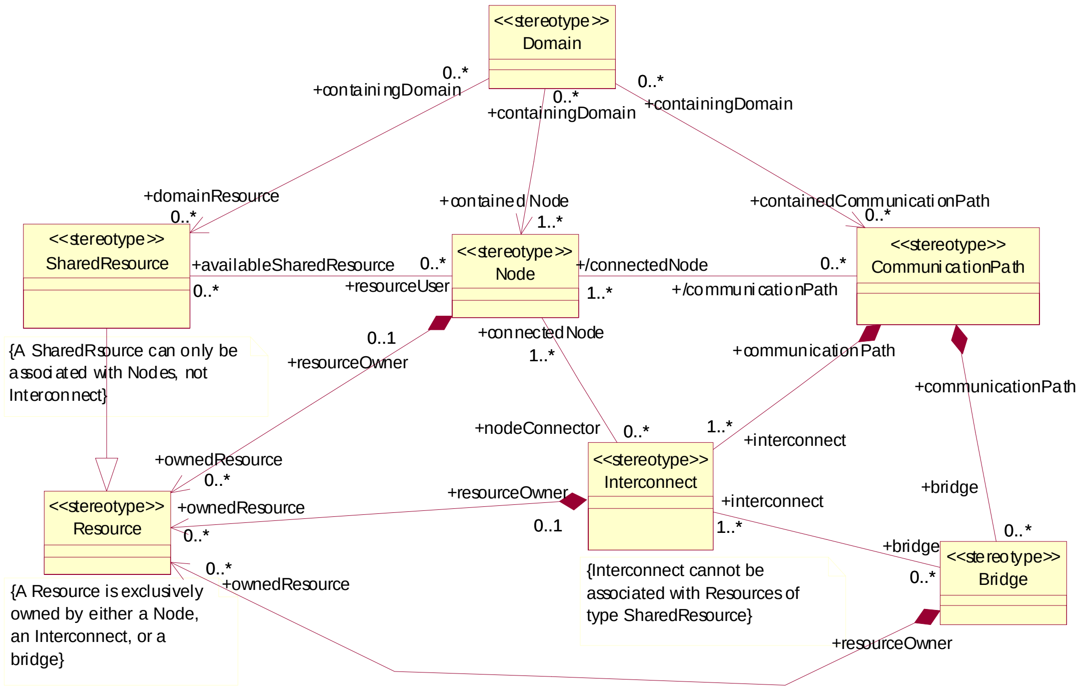


Figure 4-6 Targets package



<<stereotype>> Domain
<<stereotype>> SharedResource
<<stereotype>> Node
<<stereotype>> CommunicationPath
<<stereotype>> Resource
<<stereotype>> Interconnect
<<stereotype>> Bridge
+containingDomain
+domainResource
+containedNode
+containedCommunicationPath
+availableSharedResource
+resourceUser
+/connectedNode
+/communicationPath
+resourceOwner
+ownedResource
+connectedNode
+communicationPath
+nodeConnector
+interconnect
+bridge
0..*
1..*
0..1
{A SharedRsource can only be associated with Nodes, not Interconnect}
{A Resource is exclusively owned by either a Node, an Interconnect, or a bridge}
{Interconnect cannot be associated with Resources of type SharedResource}

Figure 4-7 Domain stereotype definition

4.3.1 Bridge (Stereotype)

Description

The Bridge stereotype extends the UML 2.0 AssociationClass metaclass (from UML2.0::AssociationClasses). A Bridge is a special type of association that connects two or more interconnects.

A Bridge exists between Interconnects to describe an indirect communication path between nodes. If a connection is to be deployed between components that are instantiated on nodes that are not directly connected, therefore requiring bridging, the connection's requirements must be satisfied by the resources of each interconnect and bridge in between.

Attributes

name: String	The bridge's name.
label: String	An optional human-readable label for this bridge.

Associations

interconnect: Interconnect [1..*]	The Interconnects that this Bridge provides connectivity to.
ownedResource: Resource [*]	Set of Resources owned by the Bridge.

communicationPath: CommunicationPath [1]	Reference the CommunicationPath the Interconnect belongs to.

Constraints

The name must be unique within the domain.

4.3.2 CommunicationPath (Stereotype)

Description

The CommunicationPath stereotype extends the UML 2.0 CommunicationPath metaclass (from UML2.0::Deployments::Nodes). A CommunicationPath connects two or more Nodes (as opposed to only two nodes for UML 2.0 Node). A CommunicationPath may be composed of one or more Interconnects and zero or more Bridges.

Attributes

No additional attributes.

Associations

interconnect: Interconnect [1..*]	Set of Interconnect contained in the CommunicationPath.
bridge: Bridge [*]	Set of Bridges contained in the CommunicationPath.
/connectedNode: Node [*]	Set of Nodes that uses the sharedResource. This association is derived from the Interconnect::connectedNode association.

4.3.3 Domain (Stereotype)

Description

The Domain stereotype extends the UML 2.0 Class metaclass (from UML2.0::CompositeStructures::StructuredClasses). A Domain is defined as a set of Nodes, CommunicationPaths, and SharedResources. In a Domain, Nodes are connected using CommunicationPaths. It represents the entire target environment.

Attributes

label: String	An optional human-readable label for the domain.
UUID: String	An optional unique identifier for this domain.

Associations

containedNode: Node [1..*]	**Node** elements that belong to the domain.
containedCommunicationPath: CommunicationPath [*]	CommunicationPaths that provide connections between nodes.
domainResource: SharedResource [*]	Shared resources that belong to the domain.

Constraints

The top-level elements in a domain all have name attributes. These names must be unique within the domain.

4.3.4 Interconnect (Stereotype)

Description

The Interconnect stereotype extends the UML 2.0 AssociationClass metaclass (from UML2.0::AssociationClasses). It establishes connection between a set of Nodes and Bridges.

An Interconnect provides a shared direct connection between one or more nodes. It can have resources, but no shared resources. Resources are matched against a connection's requirements at deployment time.

An Interconnect that is attached to only a single node can be used to describe the loopback connection. A loopback connection is implicit; components can always be interconnected locally. Sometimes, it may be useful or necessary to describe the type(s) of available loopback connections (e.g. “shared memory”), or their resources or capabilities (e.g. latency).

Attributes

name: **String**	The interconnect’s name.
label: **String**	An optional human-readable label for the interconnect.

Associations

connectedNode: **Node** [1..*]	Set of nodes that the Interconnect is connected to.
bridge: **Bridge** [*]	The bridges that provide connectivity to other interconnects.
ownedResource: **Resource** [*]	Set of Resources owned by the Interconnect.
communicationPath: **CommunicationPath** [1]	Reference the CommunicationPath the Interconnect belongs to.

Constraints

The name must be unique within the domain

4.3.5 Node (Stereotype)

Description

The Node stereotype extends the UML 2.0 Node metaclass (from UML2.0::Deployments::Nodes).

Nodes are connected to zero or more CommunicationPaths that enable components that are instantiated on this node to communicate with components on other nodes. Nodes may own resources and may have access to shared resources that are shared between nodes.

Attributes

name: String	The node's name.
label: String	An optional human readable label for the node.

Associations

nodeConnector: Interconnect [*]	Set of Interconnect to which the node is connected.
/communicationPath: CommunicationPath [*]	Set of CommunicationPath to which the node is connected. This association is derived from the Interconnect::communicationPath association.
ownedResource: Resource [*]	Set of resources owned by the Node.
availableSharedResource: SharedResource [*]	Set of SharedResources that the Node has access to.

Constraints

The name of the **Node** must be unique within the **Domain** (see above).

4.3.6 Resource (Stereotype)

Description

The Resources stereotype extends the UML 2.0 Class metaclass (from UML2.0::Kernel).

Resource represent features within the target environment. They are matched against implementation requirements at deployment planning time.

Attributes

name: String	An optional name for the requirement satisfier.
resourceType: **Sequence(String)**	The resource types that can be satisfied by this resource.

Associations

No additional associations.

Constraints

The name of a Resource must be unique within the container.
A Resource is exclusively owned by either a Node, an Interconnect, or a bridge.

4.3.7 SharedResource (Stereotype)

Description

The SharedResources stereotype extends the UML 2.0 Class metaclass (from UML2.0::Kernel). It is a specialization of the Resource stereotype.

Shared resources are resources that are shared between nodes. They are semantically equivalent to "normal" resources; however, the planner must make sure that a shared resource is not exhausted by using it from multiple nodes in parallel.

Attributes

No additional attributes.

Associations

resourceUser: **Node** [1..*]	Set of nodes that have access to the SharedResource.

Constraints

The name of the SharedResource must be unique within the domain.
A SharedRsource is a type of Resource that can only be associated with Nodes.

PSM for CCM 5

Contents

This chapter includes the following topics.

5.1 Introduction

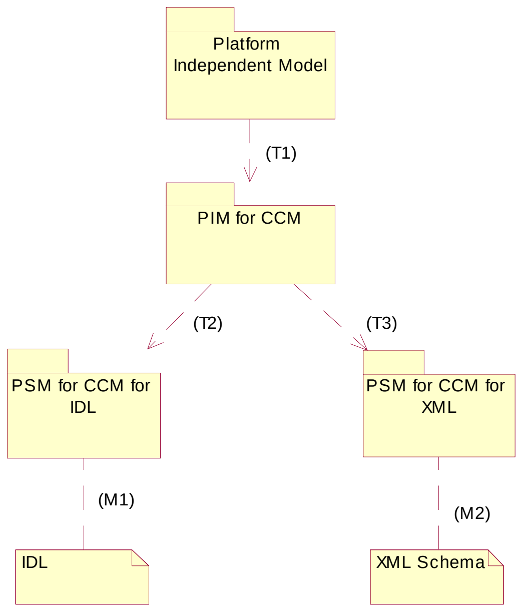
This chapter describes the mapping of the platform-independent model for Deployment and Configuration to the CORBA Component Model platform. It is intended to be a replacement for the Packaging and Deployment chapter of the CCM specification in CORBA 3.0 as well as the XML DTD chapter (chapters 6 and 7 of the latest document from the CCM RTF 1.1, ptc/02-08-03). Issues of migration and compatibility to this previous CCM deployment specification are addressed in Section 5.9, “Migration Issues,” on page 5-20.

The D&C data models are used in two different ways, first for persistent storage and distribution of information, and second for representing data at runtime. For persistent storage and distribution, the data models are mapped to XML schemas, so that information can be stored in XML files according to the model. We frequently use the term (and stereotype) *description* for the classes that define the data model. We use the term “*descriptor*” to refer to the XML file that contains the data. For runtime, the data models are mapped to IDL data structures.

The management classes are runtime entities and mapped to IDL interfaces only.

This section does not include XML schema and IDL files, since both are generated according to rules. However, these files are supplied with this specification to show the results of this rule-based file generation. The rules that will be used to auto-generate these files from the platform independent model use stereotype classes and associations appropriately and then use rules set forth in the UML profile for CORBA.

This chapter defines three transformations and two mappings.



The first transformation, T1 (*PIM* to *PIM for CCM*), takes the platform-independent model, and refines it into a platform independent model for CCM. In this *PIM for CCM*, the abstract meta-concepts are concretized, and also some other classes are aligned with the CORBA Component Model.

The second transformation T2 (*PIM for CCM* to *PSM for IDL*) takes the *PIM for CCM* and transforms it into a *PSM for CCM for IDL* that can be used to generate concrete IDL from the model. The third transformation T3 (*PIM for CCM* to *PSM for CCM for XML*) creates a *PSM for CCM for XML* that can be used to generate concrete XML schemas.

The motivation for transformations T2 and T3 is to transform the PIM into PSMs so that generic, rule-based mappings M1 and M2 can be used. (Note that some classes have different representations in IDL and XML, for example the <ClassName>Any class, prohibiting IDL and XML schema generation from the same model.) The motivation for transformation T1 is that some CCM specific transformations are necessary that are independent of the mapping to IDL or XML.

The M1 mapping is realized using the UML Profile for CORBA, the M2 mapping is defined in Chapter 6, “Mapping to XML Schema.”

5.2 Definition of Meta-Concepts

This section provides a concrete definition for the classes that are abstract in the PIM. This section is unrelated to the transformations, which will be described in the following sections.

5.2.1 Component

The abstraction of **Component** in the PIM is mapped to both components and homes for the CCM platform. Components in CCM have an interface, attributes and ports. Homes do not have ports, but an interface and attributes. Both components and homes have explicitly “supported” interfaces in addition to the “equivalent” interface, that inherits all supported interfaces, and includes attributes and explicit operations in the component and home interface definitions.

Viewing homes as a kind of component allows this specification’s model to deploy homes (by themselves or as part of an assembly). Applications or other components in an assembly can then use the home to create component instances at runtime. This supports the full feature set of CCM, without requiring explicit home implementations.

If a CCM home or component supports an interface, their **ComponentInterfaceDescription** has a special port named “**supports**” that can be used in connections for any of the “supported” interfaces. If, in an assembly, a connection is to be provided by any of the component’s or home’s supported interfaces, then the port name of the **ComponentExternalPortEndpoint** or **SubcomponentPortEndpoint** class is “**supports**.” For CCM homes, this port also provides their equivalent interface. The “**supports**” port for CCM components does *not* provide the equivalent interface, since this would be problematic for assembly implementations of components. Home implementations are always monolithic. (Note that in CCM 3.0, assemblies did not allow connections to a component’s equivalent interface either.)

Configuration properties of components, as described by the **ComponentPropertyDescription** class, are attributes in the component or home interface or any inherited component or home interface, but not in any supported interface.

Note – The "**supports**" magic name has been chosen because it reflects the supported interface. Because it is an IDL keyword, it has little likelihood of conflicting with other port names.

5.2.2 ImplementationArtifact

The meta-concept of **ImplementationArtifact** is mapped to a file accessible by URL. This PSM still treats files as opaque. Agreement between the author of an implementation and the **NodeManager** over the contents of an implementation artifact is assumed. This agreement, or "contract," is expressed in terms of execution parameters and an implementation's dependencies on resources provided by the node.

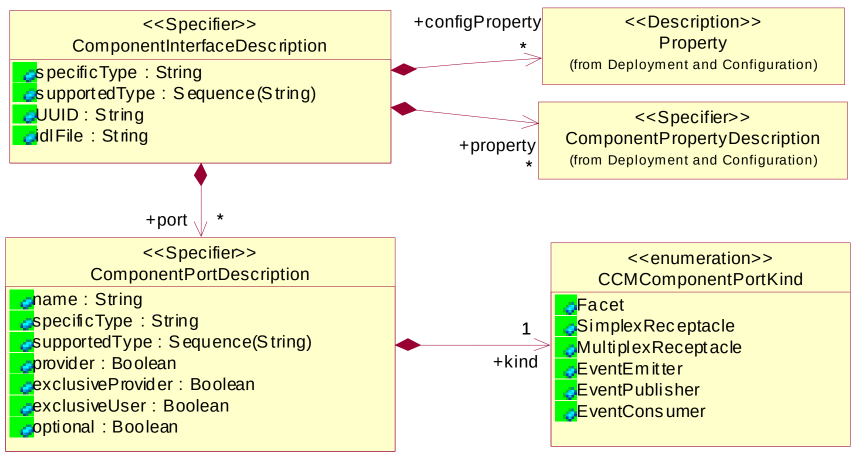
5.2.3 Package

The meta-concept of a package is mapped to a ZIP file accessible by URL, that includes implementation artifacts and descriptors. Packages have the "**.cpk**" extension and must contain a single Toplevel Package Descriptor containing a <ClassName>ToplevelPackageDescription element with the magic name "**package.pcd**."

5.3 PIM to PIM for CCM Transformation

This section defines transformation T1 (as described in the introduction for this chapter). It takes the platform-independent model from Chapter 2 and aligns classes with the CORBA Component Model. This involves changes to attributes, associations and semantics of some classes. All classes from the PIM that are not refined here are imported into the PIM for CCM without change.

5.3.1 ComponentInterfaceDescription



The **ComponentInterfaceDescription** and **ComponentPortDescription** classes are augmented to support CCM.

The **idlFile** attribute is added to the **ComponentInterfaceDescription**. If it is not the empty string, it contains a URL that references an IDL file that contains the definition for this component or home. The IDL file is not used within the deployment infrastructure; it may be included in a package for convenience. Since deployable applications have a component interface, some tools that deploy and execute such applications might need the IDL to interact with the ports of the application's component interface.

The `kind` attribute is added to the **ComponentPortDescription** class and specifies the concrete port kind that is used. This information is required by the **NodeManager** and by assembly tools. In CCM, **EventConsumer** and **Facet** ports are considered providers, the other ports are users.

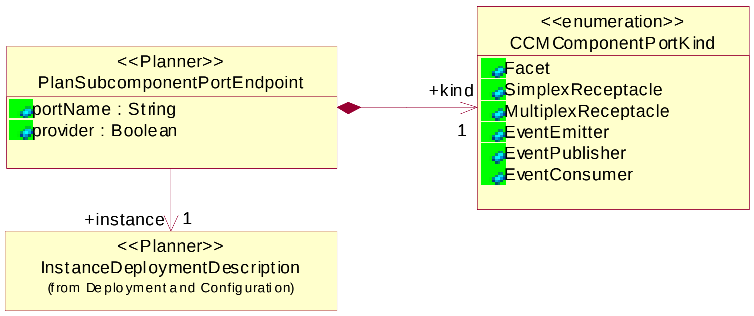
Repository Id strings are used to identify interface types, i.e. for the **specificType** and **supportedType** attributes.

For **Facet** ports, **supportedType** lists the Repository Id of the provided interface and any of its base interfaces that the developer (or tool) chooses to expose for connections. For receptacles, **supportedType** lists the Repository Id of the accepted interface. For **EventEmitter** and **EventPublisher** ports, **supportedType** lists the Repository Id of the accepted consumer interface. For **EventConsumer** ports, **supportedType** lists the Repository Id of the consumer interface and any of its base interfaces that the developer (or tool) chooses to expose for connections.

If the component or home supports one or more interfaces, this will be reflected by a **ComponentPortDescription** element of kind Facet with the magic name "**supports**." The **specificType** attribute is left empty, the **supportedType** attribute lists the Repository Id of any of its supported interfaces and base interfaces that the developer wants to expose for connections.

Initially, a **ComponentInterfaceDescription** can be generated from a component's or home's IDL description with a defined set of configuration properties (from attributes) and default values for the **exclusiveProvider**, **exclusiveUser** and **optional** attributes. If desired, a user can then adjust these three attributes for each port and also add configuration property default values to the **ComponentInterfaceDescription** by adding **Property** elements to the **configProperties** list.

5.3.2 *PlanSubcomponentPortEndpoint*



The **kind** attribute augments the `provider` attribute in the **PlanSubcomponentPortEndpoint** class and specifies the concrete port kind that is used. This information is required by the various managers in the Execution Management Model. The **provider** attribute still indicates a port which provides an object reference.

5.3.3 *Application*

The **start** operation on the **Application** class performs the **configuration_complete** operation in all component instances that are part of the application.

5.3.4 *RepositoryManager*

When artifact files are included in the package (as opposed to referenced via URL outside the package), the **RepositoryManager** must make its own copy of these artifacts during the **installPackage** operation. It must substitute an URL that references this copy of the artifact in the **location** attribute of **ImplementationArtifactDescription** elements delivered via its interface.

5.3.5 *SatisfierProperty*

This PSM has to define concrete types that are implied on the value of a **SatisfierProperty** by the **SatisfierPropertyKind**, and on the value of the **Property** that is matched against the satisfier.

- For the **Quantity** kind, the value of the **SatisfierProperty** is of type **unsigned long**. The value of the **Property** is ignored.

- For the **Capacity** kind, the value of the **SatisfierProperty** is of type **unsigned long** or **double**. The value of the **Property** must be of the same type.
- For the **Maximum** and **Minimum** kinds, the value of the **SatisfierProperty** is of type **long** or **double**. The value of the **Property** must be of the same type.
- For the **Attribute** kind, the value of the **SatisfierProperty** is of type **long**, **double**, **string**, or an enumeration type. In the case of long, double or string, the value of the **Property** must be of the same type. If the value of the **SatisfierProperty** is of enumeration type, the value of the **Property** is of type **string**, containing the enumeration value that must compare equal to the **SatisfierProperty** value.
- For the **Selection** kind, the value of the **SatisfierProperty** is a sequence of type **long**, **double**, **string**, or an enumeration type. The same rules as for the **Attribute** kind apply.

5.4 PIM for CCM to PSM for CCM for IDL Transformation

This section defines transformation T2 (as described in the introduction). It transforms the *PIM for CCM* into a *PSM for CCM for IDL* that can be used to generate concrete IDL using a rule-based mapping. Classes from the *PIM for CCM* are transformed to match the UML Profile for CORBA. Its rules are then used to generate concrete IDL.

The first subsection describes generic mapping rules that are applied to all classes that are part of the *PIM for CCM*. The second subsection defines special transformation rules for the classes that are abstract in the PIM.

All classes in the *PSM for CCM for IDL* are placed in the **Deployment** package, so that all resulting IDL structures and interfaces will be part of the `Deployment` IDL module.

5.4.1 Generic Transformation Rules

The mapping to IDL is accomplished using the rules set forth in the UML Profile for CORBA. To apply these rules, the stereotypes used in the platform-independent model are mapped to stereotypes for which a mapping is defined in the profile. The «**Description**» stereotype and all that inherit from it are mapped to the **«CORBAStruct»** stereotype; these classes are therefore mapped to CORBA structures. The «**Exception**» stereotype is mapped to the **«CORBAException»** stereotype; such classes become CORBA exceptions. The «**Enumeration**» stereotype is mapped to the **«CORBAEnum»** stereotype in order to become enum types in IDL. The «**Manager**» stereotype is mapped to the **«CORBAInterface»** stereotype so that these classes become CORBA interfaces.

To avoid redundancy and circular graphs, non-composite associations between classes with a common owner are expressed by an ordinal attribute at the source (navigating) end, with the name of the attribute being the role name plus the suffix "**Ref**," and the type "**unsigned long**." The value of this attribute is the index of the target element in its container. To enable the usage of an index, the composition of the target element in its container is qualified with the "ordered" constraint.

Wherever the multiplicity at the navigable end of a navigable association or composition is not exactly one (but 0..1, 1..* or *), a new UML class is introduced to represent a sequence of the class at the navigable end of the association or composition. The sequence class has the «**CORBASequence**» stereotype, and the name is the english plural of the name of the element class. The sequence class has a composition association with the element class that is navigable from the sequence to the element. The composition is qualified with the index of the sequence and the multiplicity of the original association. The original association or composition is then replaced with an association or composition with the original role name to the new sequence class, with a multiplicity of 1 at the navigable end. According to the rules in the UML Profile for CORBA, these associations will then map to a structure member in IDL, its type being a named sequence of the referenced type.

A similar rule is applied for all uses of the **Sequence** data type, which is used to denote sequences of elements of the same type that are to be passed to an operation as a single parameter, or returned from an operation as a single return value. (With the exception of a sequence of strings, which re-uses the **StringSeq** type in the **CORBA** package, see below.) In these cases, a new class is introduced to represent a sequence of the contained type as above. The parameter or return value will then use the new class.

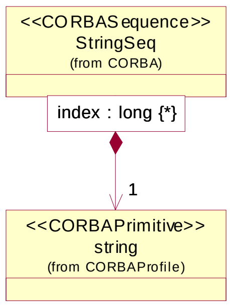
Note that in combination, these rules map non-composite associations between classes with a common owner and a multiplicity other than 1 to sequence of "**unsigned long**" type.

The inheritance relationships of classes with the «**Description**» stereotype (**SharedResource**, **Resource** and **Capability**) classes are removed; all attributes and associations of the base class are attached to the derived class.

Associations of classes with the «**Manager**» stereotype are removed from the *PSM for CCM for IDL*.

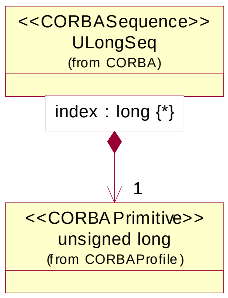
5.4.2 Special Transformation Rules

5.4.2.1 Sequence(String)



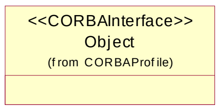
A type representing a sequence of strings already exists in the **CORBA** package and can be re-used. The **Sequence(String)** type is therefore mapped to the **StringSeq** class from the CORBA package as shown above. It then maps to the **CORBA::StringSeq** type in IDL (from the **orb.idl** file).

5.4.2.2 Sequence(unsigned long)



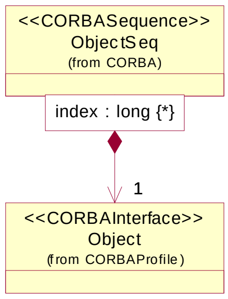
A type representing a sequence of the `unsigned long` type already exists in the **CORBA** package and can be re-used. The **Sequence(unsigned long)** type is therefore mapped to the **ULongSeqSeq** class from the CORBA package as shown above. It then maps to the **CORBA::ULongSeq** type in IDL (from the **orb.idl** file). Sequences of the unsigned long type occur when a non-composite association between classes with a common owner with a multiplicity other than one occurs, according to the generic rule above.

5.4.2.3 Endpoint



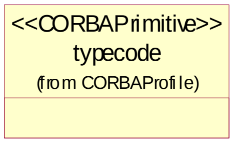
The abstract **Endpoint** class is mapped to the **Object** class from the **CORBAProfile** package. It will therefore map to the **Object** type in IDL.

5.4.2.4 Sequence(Endpoint)



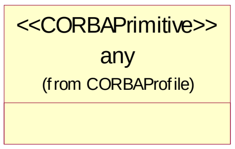
A type representing a sequence of object references already exists in the **CORBA** package and can be re-used. The **Sequence(Endpoint)** type is therefore mapped to the **ObjectSeq** class from the CORBA package as shown above. It then maps to the **CORBA::ObjectSeq** type in IDL (from the **orb.idl** file).

5.4.2.5 DataType



The abstract **DataType** class is mapped to the **typecode** class from the **CORBAProfile** package. It then maps to the **TypeCode** type in IDL.

5.4.2.6 Any



The abstract **Any** class is mapped to the **any** class from the **CORBAProfile** package. It will then map to the **any** type in IDL.

5.4.2.7 Primitive Types

The UML data types **String**, **Integer** and **Boolean** are mapped to the classes **string**, **long** and **boolean** in the **CORBAProfile** package, respectively. They will then map to the `string`, `long` and `boolean` types in IDL, respectively.

5.4.3 Mapping to IDL

After applying the transformations defined in this section, IDL is generated by applying the rules set forth in the UML Profile for CORBA specification.

Note – Insert reference. Put IDL into Appendix and cross-link here.

5.5 PIM for CCM to PSM for CCM for XML Transformation

This section defines transformation T3 (as described in the introduction). It transforms the *PIM for CCM* into a *PSM for CCM for XML* that can be used to generate concrete XML schema using a rule-based mapping.

5.5.1 Generic Transformation Rules

Data model elements, annotated with the «**Description**» or «enumeration» stereotype (or a stereotype that inherits from it), are used to generate XML schema for persistent storage of metadata. Management model elements, annotated with the «**Manager**» or «**Exception**» stereotype, are only mapped to IDL, but not to XML.

All mapping rules can be implemented by a transformation based on the XMI representation of the platform independent model.

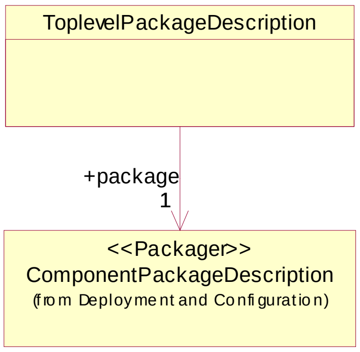
Certain classes (and their contained associations) will be mapped into XML files, which we call descriptors, and XML elements within those files. Composition associations imply that the class at the part end is in the same XML descriptor file as the class at the composite (containing) end. Non-composite associations between classes with a common owner (composite end of composition) are implemented by an identifier attribute at the target and a matching reference attribute at the source. Both attributes are of type **String**. The attribute at the target of navigation has the name "**id**," the attribute at the navigating end uses the role name plus the suffix "**Ref**." The **"id"** attributes are scoped by the common owner.

Non-composite associations between classes that do not have a common owner are mapped to two optional attributes "**fileinarchive**" and "**link**," both of type **String**, with an exclusive-or relationship between them, so either the one or the other must be present. The **fileinarchive** attribute points to another XML descriptor file within the same package. The `link` attribute contains a URL that points to the location of another descriptor file. This type of association only appears in the Component Data Model. All classes in the Target Data Model have the **Domain** as a common owner, all classes in the Execution Data Model have the **DeploymentPlan** as a common owner.

Inheritance relationships (used by the **SharedResource**, **Resource** and **Capability** classes) are removed; all attributes and associations of the base class are attached to the derived class.

5.5.2 *Special Transformation Rules*

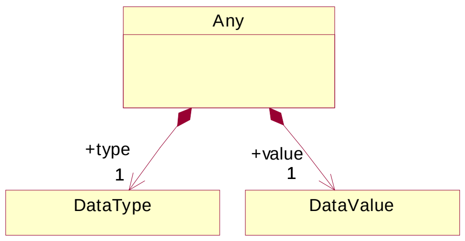
5.5.2.1 *ToplevelPackageDescription*


ToplevelPackageDescription
+package
1
<<Package>>
ComponentPackageDescription
(from Deployment and Configuration)

The <ClassName>ToplevelPackageDescription is introduced to point to the **ComponentPackageDescription** element for the top-level component package in a package.

The motivation for this element is that a package may include component packages for sub-components. A selection mechanism is necessary to distinguish the top-level component package. This is accomplished by including a single Toplevel Package Descriptor with the magic name "**package.pcd**" into the package.

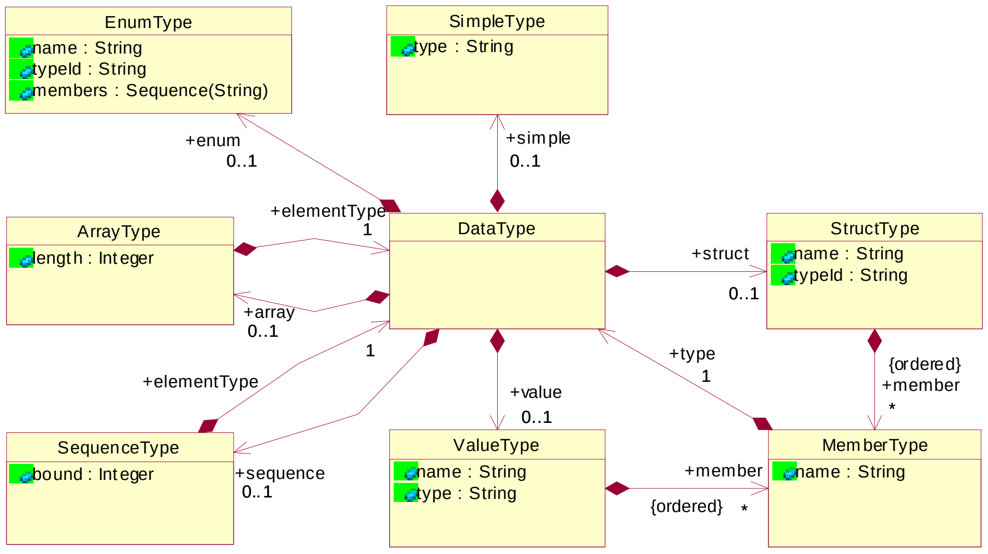
5.5.2.2 *Any*



An **Any** instance describes a typed value. It is mapped to a class that contains a <ClassName>DataType and a <ClassName>DataValue, which are elaborated below.

Note – Investigate whether the Any type that is part of XMI is sufficient.

5.5.2.3 *DataType*



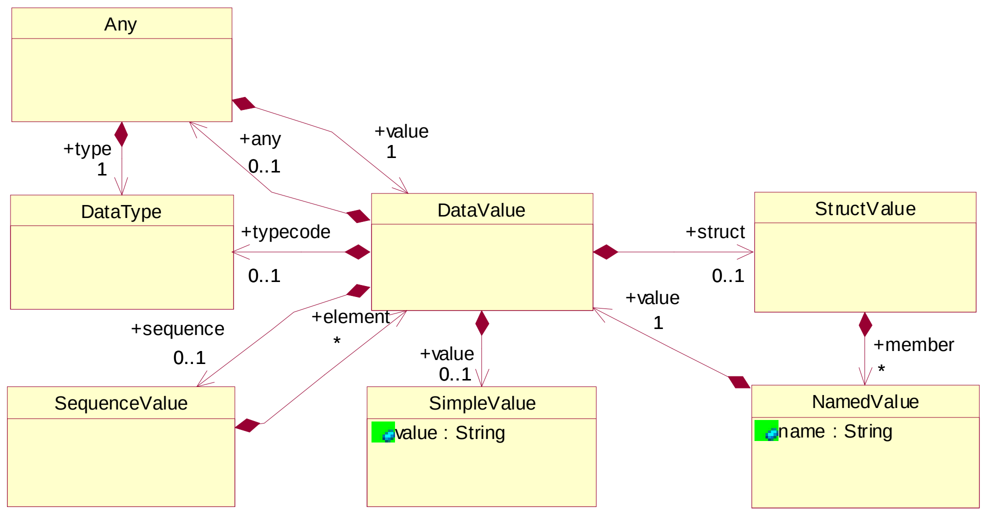
A <ClassName>DataType instance describes a type. It is mapped to a hierarchical structure as shown above, describing available types in IDL.

The <ClassName>DataType class contains a single element, either a **StructType**, **ValueType**, **SequenceType**, **ArrayType**, **EnumType** or **SimpleType**. The **SimpleType** element is used for primitive IDL types, including the **any**, **TypeCode** and **Object** types. Its **type** attribute contains the IDL name of the primitive type.

In **StructType**, **ValueType** and **EnumType**, the name attribute contains the name of the **struct**, **valuetype** or **enum** IDL type, and the **typeId** attribute contains its Repository Id.

Note – union types cannot be described using the <ClassName>DataType class above. Therefore, **union** properties cannot be configured.

5.5.2.4 DataValue



A <ClassName>DataValue instance describes a value. It is mapped to a hierarchical structure as above, fully describing a value that can be described by an IDL type. The **StructValue** element is used for both **struct** and **valuetype** values. The **SequenceValue** element is used for both **sequence** and **array** values. The **SimpleValue** element is used for primitive values, `enum` values and object references. Its **value** attribute contains a stringified representation of the primitive value. For enumeration types, this is the name of the value. For object references, the `value` attribute holds a stringified object reference. For the **octet** type, the `value` attribute holds an integer value between 0 and 255. For **char**, **wchar**, **string**, **wstring** types, the `value` attribute holds an unquoted literal (i.e., no single or double quotes, and no leading L prefix) that conforms to the IDL syntax and semantics chapter. For integer and floating point types, the **value** attribute holds an integer or floating point literal, respectively. For convenience, if the data type is a sequence or array of **octet**, the value is represented by a single **SimpleValue** element that holds, in the **value** attribute, the data in Base64 encoding.

5.5.2.5 Others

The **PackageConfiguration**, **DomainUpdateKind**, **Connection** and **Endpoint** classes are used by the runtime models only and are not part of the PSM for XML.

5.5.3 Transformation Exceptions and Extensions

The following file name extensions are used for XML descriptor files:

- The **ComponentPackageDescription** element maps to a Component Package Descriptor file with the "**.cpd**" file extension.
- The **ComponentImplementationDescription** element maps to a Component Implementation Descriptor file with the "**.cid**" file extension.
- The **ImplementationArtifactDescription** element maps to an Implementation Artifact Descriptor file with the "**.iad**" file extension
- The **ComponentInterfaceDescription** element maps to a Component Interface Descriptor file with the "**.ccd**" (CORBA Component Descriptor) file extension.
- The **Domain** element maps to a Domain Descriptor file with the "**.cdd**" (Component Domain Descriptor) file extension.
- The **DeploymentPlan** element maps to a Deployment Plan Descriptor with the "**.cdp**" (Component Deployment Plan) file extension.
- The **ToplevelPackageDescription** element maps to a Toplevel Package Descriptor with the "**package.pcd**" file name.
- Package files use the "**.cpk**" extension.

As described above, associations that cross file boundaries are mapped to the two attributes "**fileinarchive**" and "**link**." In the case of the association between **SubcomponentInstantiationDescription** and **ComponentPackageDescription**, the referenced file can be either a Component Package Descriptor or a package (i.e., a ZIP file with the "**.cpk**" extension containing the package).

For backward compatibility, if the target of a **fileinarchive** attribute that references metadata descriptors (rather than implementation artifacts) does not exist, it will also be looked for under a top level "**meta-inf**" directory.

5.5.4 Mapping to XML

After applying the transformations defined in this section, XML Schema are generated by applying the rules set forth in Chapter 6, Mapping to XML Schema.

Note – Put schema into Appendix and cross-link here.

5.6 Mapping Discussion

This section elaborates the application of the above rules to the various model segments. As the transformation and mapping rules are fully defined above, this section is not normative.

5.6.1 Component Data Model

5.6.1.1 Mapping to XML

The Component Data Model will be mapped to several XML schemas as indicated by non-composite associations. Instances of the model (XML files) will describe pieces of an implementation relating to the actors that handle the data.

One schema will describe the Component Interface Descriptor containing data from the **ComponentInterfaceDescription**. The Specifier will create this file along with the IDL for the component interface. A Component Interface Descriptor can initially be auto-generated from the component's or home's IDL; some parts can later be edited (such as the default values of configuration properties).

The Component Implementation Descriptor describes a single implementation and is created by the Developer that creates a monolithic implementation or by the Assembler that creates an assembly-based implementation. It contains information from the **ComponentImplementationDescription**, the **ComponentAssemblyDescription** and the **MonolithicImplementationDescription.**

The Implementation Artifact Descriptor contains information from the **ImplementationArtifactDescription** and is created by the supplier of that implementation artifact (the Developer or an external source). Note that the **ImplementationArtifactDescription** elements directly contained by the **MonolithicImplementationDescription** (the "primary artifacts") are not in a separate descriptor file, while any secondary artifacts on which the first artifact depends (such as ORB dlls) will have their own descriptor file.

The Component Package Descriptor contains data from the **ComponentPackageDescription**. It is created by the Packager that packages up one or more implementations.

Note – Administrators might want to disallow external packages that cannot be validated. Whether a Repository Manager provides the option to disallow such "**link**" attribute in a package or not is a quality of implementation issue.

5.6.1.2 Mapping to IDL

All classes in the Component Data Model are mapped to CORBA structures in the Deployment module using the generic mapping rules. No exceptions are necessary.

5.6.2 Component Management Model

The **RepositoryManager** is mapped to a CORBA interface in the Deployment module using the generic mapping rules. Since it has the «**Manager**» stereotype, it is not mapped to XML.

5.6.3 Target Data Model

5.6.3.1 Mapping to XML

All classes in the Target Data Model are mapped to IDL and XML using the generic mappings, resulting in a single XML schema. Consequently, domain information will be contained in a single XML file. It could be argued that there is no need for an XML mapping of the Target Data Model. If the target information is created using a proprietary tool that comes with the **TargetManager**, using a proprietary means for feeding that information to the **TargetManager** would be comformant. However, the XML mapping comes at no price, so it is included to discourage incompatible XML schemas.

5.6.3.2 Mapping to IDL

All classes in the Target Data Model are mapped to CORBA structures in the Deployment module using the generic mapping rules.

5.6.4 Target Management Model

The **TargetManager** is mapped to a CORBA interface in the Deployment module using the generic mapping rules. Since it has the «**Manager**» stereotype, it is not mapped to XML.

The **DomainUpdateKind** class is used at runtime only. It is mapped to IDL using the generic mapping rules, but not mapped to XML.

5.6.5 Execution Data Model

5.6.5.1 Mapping to XML

The **DeploymentPlan** is mapped to a single XML schema. Consequently, a concrete **DeploymentPlan** will be contained in a single XML file.

5.6.5.2 Mapping to IDL

All classes in the Execution Data Model are mapped to CORBA structures in the Deployment module using the generic mapping rules.

5.6.6 Execution Management Model

All classes in the Execution Management Model are mapped to CORBA interfaces in the Deployment module using the generic mapping rules. Since they have the «**Manager**» stereotype, they are not mapped to XML.

5.7 Miscellaneous

5.7.1 Entry Points

CCM's Packaging and Deployment chapter in CORBA 3.0 defines a home factory entry point that enables a container to create a user-defined home using a user-defined factory.

This specification defines the interaction between an implementation artifact and the execution manager as implementation-dependent, in order to not restrict the forms that an implementation artifact might have – executable files, loadable libraries, source files or scripts, for example.

However, to ensure source code compatibility in the common case without restricting implementation choice, entry points are defined here if the language is C++ and the implementation artifact is a shared library, or if the language is Java and the implementation artifact is a class file. In these two cases, there must be a specific execution parameter associated with the Monolithic Implementation Description.

If the instance to be deployed is a component, then the name of the execution parameter shall be "**component factory**." The parameter is of type **String**, and its name is the name of an entry point that has no parameters and that returns a pointer of type **Components::EnterpriseComponent**.

If the instance to be deployed is a home, then the name of the execution parameter shall be "home factory." The parameter is of type **String**, and its name is the name of an entry point that has no parameters and that returns a pointer of type **Components::HomeExecutorBase**.

For backwards compatibility, it is recommended that the name of the entry point should be the name of the component or home, prefixed with "**create_**" (e.g. "**create_Account**" for an Account component).

If the language is C++, then the entry points shall be qualified as '**extern "C"**'.

These well-defined entry points ensure that the user code for the entry point does not need to be changed when building components for different target environments. These definitions do not enable interoperability between containers and DLLs (even assuming the same compiler and ORB), thus additional interfaces are still required that are specific to container implementations. This implies that, as in CCM 3.0, component and home implementation DLLs are specific to the container implementation (and the code generation tools). Since there was and is no normative interoperability interfaces within a node, thus further implies that there is no vendor segmentation boundary within a node at all.

5.7.2 Homes

Note that this specification does not depend on the existence of homes; using the entry points defined above, a container is able to create component instances directly, without the need of creating a home first, and then using it as a factory for the component instance.

This is no loss in comparison to the Packaging and Deployment chapter of CCM in CORBA 3.0. If a component instance is to be deployed as part of an assembly, the container has no way of providing a user-defined home with any parameters, and is therefore limited to keyless homes. However, a factory operation for the component instance as defined above can do its job as well as the parameter-challenged `create` operation that is part of a keyless home.

In contrast to the Packaging and Deployment chapter, this specification recognizes homes as instances that can be deployed, and therefore enables the full range of home features.

5.7.3 Valuetype Factories

If an **ImplementationArtifact** contains valuetype factories, then its list of execution parameters shall include an element with the name “valuetype factories” and of type **ValuetypeFactoryList**, which is defined as

```
module Deployment {
    struct ValuetypeFactory {
        string repid;
        string valueentrypoint;
        string factoryentrypoint;
    };
    typedef sequence<ValuetypeFactory>
        ValuetypeFactoryList;
};
```

Each element of that sequence describes a valuetype factory that needs to be registered with the ORB in order to demarshal user-defined valuetypes. The `repid` field specifies the Repository Id of the valuetype created by the valuetype factory. The `factoryentrypoint` field specifies the name of an entry point that can be be used to create an instance of the valuetype factory. If **valueentrypoint** is not the empty string, it specifies an entry point that can be used to create an instance of the valuetype.

If the language is C++, then the entry points shall be qualified as ‘**extern** `"C"`’.

5.7.4 Discovery and Initialization

The **ExecutionManager** must be able to find the **NodeManager** instances for all nodes in the **Domain**, so that it is able to deploy applications according to deployment plans that are based on the current contents of the Target Data Model. This is accomplished using the Naming Service.

- The user of the deployment system creates a naming context for a domain. Note that a naming context is expressible by a URL representation (e.g., a "**corbaname:**" reference).
- Implementations of the **ExecutionManager** interface must accept the address of the naming context as a configuration parameter, and use it to publish its own reference with the name "**ExecutionManager**" and the empty string as the id in that context.
- Implementations of the **TargetManager** interface must accept the address of the naming context as a configuration parameter, and use it to publish its own reference with the name "**TargetManager**" and the empty string as the id in that context.
- Implementations of the **NodeManager** interface must accept the address of the naming context as a configuration parameter, and use it to publish their own reference with the node's name as the name and the id "**NodeManager.**" The node's name must match the name attribute of the node in the Target Data Model.

Upon startup, the **ExecutionManager** finds the **TargetManager** in the Naming Service, and accesses the current **Domain** information. Based on the **Node** elements that are contained in the **Domain**, the **ExecutionManager** then calls the **joinDomain** operation of each **NodeManager.**

An **ExecutionManager** may offer functionality to "add" new nodes to the domain, or to remove nodes from the domain. In that case, the **ExecutionManager** looks up a **NodeManager** with a user-provided name in the Naming Service and then calls its **joinDomain** or **leaveDomain** operation, respectively. In addition, an **ExecutionManager** may offer to scan the Naming Service context for previously unregistered nodes, calling the **joinDomain** operation on each associated **NodeManager.**

Note that there is no direct relationship between domains and repositories. Therefore, implementations of the **RepositoryManager** interface are not registered in the Naming Service.

5.7.5 Location

URL references are handled by the **RepositoryManager** and **NodeManager** interfaces: the **RepositoryManager** receives URLs to packages as a parameter to the **installPackage** operation and must generate URLs pointing to itself in **ImplementationArtifactDescription** elements. The **NodeManager** receives URLs as attributes of the **ArtifactDeploymentDescription** elements that are part of the **DeploymentPlan**.

Both **RepositoryManager** and **NodeManager** shall be able to interpret URLs according to the **http** scheme. Additional schemes may optionally be supported.

Note – This requires **RepositoryManager** implementations to include both an **http** server and an **http** client. **NodeManager** need to implement **http** clients only, in order to download implementation artifacts from the repository.

If a **RepositoryManager** supports URL schemes in addition to http, it shall offer a configuration parameter that allows user selection of the scheme(s) that will be used in **ImplementationArtifactDescription** elements.

5.7.6 Segmentation

This specification obsoletes CCM's idea of component segmentation. In the original CCM, assemblies provided just a single level of decomposition. Segments then offered a second level to split the implementation of a component into several independent pieces of code. This specification allows composition and decomposition on any level, and therefore the ability to add another level of decomposition on the lowest level is redundant. However, no parts of this specification inhibit a component author from using this feature of the CCM Implementation Framework.

5.8 Impact on the CCM Specification

This specification is intended to replace the Packaging and Deployment chapter and the XML DTD chapter of CCM 3.0.

Note – The Packaging and Deployment chapter of CCM 3.0, in its Component Deployment section, defines interfaces that are involved in the deployment of components onto nodes. Similar interfaces might be useful in implementing the NodeManager, however, this specification does not prescribe any such node-level interfaces.

The potential ability to create component instances without homes requires that the **get_ccm_home** operation in the **CCMObject** interface is allowed to return a nil object reference.

5.9 Migration Issues

This section deals with the issues of migrating from the Packaging and Deployment model that exists in CCM 3.0 to the deployment model presented in this specification.

5.9.1 Component Implementations

The portable parts of CCM component implementation source code remains untouched. The generated code to enable interactions with the containers may change, requiring recompilation and linking. The non-portable hand written code in some implementations which was written assuming a particular container implementation would likely have to change — similar to porting the component to a different CCM system.

5.9.2 Component and Assembly Packages and Metadata

The metadata is changed to be based on XML schemas, and the basic models are different. Many lower level elements are not different, and it is expected that meta-data transformation (forward migration) will be able to be automated in the common cases where all the features used are supported.

This specification is kept simple in anticipation that broad (and necessarily complex) software packaging and distribution standards do not exist, and the W3C OSD specification (by Microsoft and Marinba in 1997) referenced by the original CCM specification did not become a standard. Future RFPs may want to consider mappings from such comprehensive standards into this simpler model that focuses on CCM applications.

The component data model stays within the scope of deployment and configuration and does not bring forward all the metadata aspects in the previous CCM specification that were not relevant to deployment and configuration. Furthermore, much of the metadata for informing containers of the requirements of component instances was not defined as part of an intervendor boundary. Thus this specification assumes the use of two "private" channels of information between the development tools (and code generation) and the runtime environment (NodeManager). These are the resource requirements of the MonolithicImplementationDescription and the execParameters of the InplementationArtifactDescription. The submitters believe standardizing this metadata should be part of a true effort at vendor segmentation between CCM development tools and CCM runtime environments (assuming the same compiler and ORB), which does not exist and was not the mandate of this RFP.

Beyond the necessity of validating configuration and connection among components, the one other metadata interoperability issue is to standardize the vocabulary for selection criteria, which is interoperation between users and implementers of component software. This is currently deferred due to the concurrence of the other specification for this language with this specification (see below).

5.9.3 Component Deployment Systems

Deployment systems need to be changed to support this specification. Most aspects of container implementations should be reusable.

5.10 Metadata Vocabulary

5.10.1 Implementation Selection Requirements

Selection requirements, part of both the **PackageConfiguration** and **SubcomponentInstantiationDescription** classes, express requirements that are meant to drive the selection among alternative implementations. The user of an implementation (creator of a package configuration or an assembly) is requesting services to be satisfied by a component implementation. The mechanism defined in this specification requires agreement of the vocabulary of these services on both sides, but there is no interoperable vocabulary defined. The currently active RFP entitled "UML Profile for

Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms" should result in, among other things, "a Definition of Individual QoS Characteristics," which should provide an appropriate vocabulary to drive this mechanism.

When this QoS-driven vocabulary is connected to the CCM PSM, some other component metadata requirements, such as "humanlanguage" may also be added to the selection criteria language.

5.10.2 Monolithic Implementation Resource Requirements

As mentioned above, this vocabulary is a private communication channel between development tools and the NodeManager, since no other interoperability boundary exists between these two. Obviously some standardization could be easily done, based on previous CCM-defined metadata such as container supported persistence, transactions, and POA policies. If this limited scoping is not accepted by the Task Force, data model classes containing this type of information can easily be added to support both a defined resource vocabulary and even a separate container-services vocabulary for information that would never be part of a "resource finding" matching process with the target nodes, but needs to be conveyed to the runtime environment for component instances.

Mapping to XML Schema 6

XML documents are an attractive and widely used format for various descriptors defined by models. The instantiation of these XML documents is guided by one or more XML Schemata, derived from the PIM constructs through a mapping process.

The mapping rules for the XML Schemata have been derived from, and are compatible with, the XML Metadata Interchange (XMI) specification, version 2.0. However, the information to be mapped in this case are not fully featured metamodels, but data structure definitions (see below), the extra capabilities provided by the XMI schema are not needed. To simplify the use of the resulting XML schema in a resource constraint environment, where a MOF or XMI tools are most likely not available, the mapping has been constructed using plain XML Schema only, and the XMI schema is not imported.

As described earlier in this document, the PIM is segmented into a set of sub-models. Along this line is again a distinction between management and data models. The data models define the actual information required for the deployment process and are reflected in the XML-encoded information sets presented to the deployment system. Therefore only the three data sub-models of the PIM are mapped to XML Schemata, which are the Component, Target and Execution Data Models.

Common mapping rules have been applied to all three models. Only the mapping of links has variations, as described later. XML Schemata, as the successors of DTDs, are typically used for validation of XML documents. The XML Schemata presented here are intended to go beyond that role and are capable to assist the generation and interpretation of deployment data structures.

Two namespaces are used in the resulting XML Schema: Namespace "xsd" is used for all elements defined by the XML Schema metaschema, while "DnC" is the (default) target namespace for all elements defined in the created schema. It is explicitly used in all type references.

Classes in the PIM map to complexType definitions in the XML Schema. XMI provides two alternatives to map UML attributes: either as XML attributes or as XML elements. In this mapping, only elements are used. The disadvantage of an increased XML document size for this method is more than compensated by the gain in flexibility for the value encoding. The element method is more tolerant to the used character set and in particular immune to embedded whitespace.

The definition of type Any as a catch-all type has been borrowed from the XMI v2.0 specification:

```
<xsd:complexType name="Any">
  <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:any processContents="skip" />
  </xsd:choice>
  <xsd:anyAttribute processContents="skip" />
</xsd:complexType>
```

Regular associations in the UML models (not aggreagations) are mapped to XLinks in the XML Schema. Unidirectional associations result in a single link following the navigable direction of the association. Bidirectional associations result in two complementary links. Names for the linking elements are derived from the association and aggregation role names in the UML model.

Real XLinks, which may cross file boundaries without problems, are only used in the Component and Target Data Model. According to the view of the Deployment Plan as a flattened structure containing all relevant information, more compact IDREF links are used in the schema derived from the Execution Data Model. Due to the fattened character of the Deployment Plan, the actual linking takes place when the individual elements are integrated into the plan. To accommodate this, the XML type DeploymentPlan uses sub-classing to generate the link targets required for the IDREF links. This is shown in the following excerpt from the DeploymentPlan XML type:

```
<xsd:element name="artifact">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="DnC:ArtifactDeploymentDescription">
        <attribute name="ïd" type="xsd:string" use="required" />
      </xsd:extension>
    </xsd:complexContent>
  </complexType>
</xsd:element>
```

Here the ArtifactDeploymentDescription XML type is extended with a mandatory attribute when integrated into the deployment plan. This attribute provides the target id for an IDREF link, which is referenced like this:

```
 <xsd:element name="implementingArtifact">
   <!-- Reference to Artifact -->
   <attribute name="ïdref" type="xsd:IDREF" use="required" />
 </xsd:element>
```

In the resulting Deployment Plan XML document this results in the following snippets:

Defined as:

```
...
<artifact id="1234">
   ...
</artifact>
```

and referenced via:

```
<implementingArtifact idref="1234">
```

Elements with multiplicity greater than one are mapped into "choice" content groups if no ordering is required. The "sequence" content group is used for order-sensitive elements. XOR constraints map to a "choice" group with an upper limit of 1.

According to the following snippet, the "property" subelement may be absent from the body of the instantiated "requirement" element, or repeated an unlimited number of times:

```
  <xsd:complexType name="Requirement">
    <xsd:sequence>
      <xsd:element name="resourceType" type="xsd:string" />
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="property" type="DnC:Property" />
      </xsd:choice>
    </xsd:sequence>
</xsd:complexType>
```

In the resulting XML document, a conforming instance could look like the following snippet:

```
 <myRequirement>
   <resourceType>someResource</resourceType>
   <property>
     ...
   </property>
   <property>
     ...
   </property>
 </myRequiremetn>
```

Generalization in the UML model is mapped to the "extends" single inheritance construct of XML Schema. In the following snippet is Resource specialized by SharedResource:

```
<xsd:complexType name="SharedResource">
  <xsd:extension base="DnC:Resource">
    <xsd:complexContent>
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="node">
          <xsd:complexType>
            <!-- Hosting Node pointer -->
            <xsd:attribute name="href" type="xsd:string"
                           use="required">
          </xsd:complexType>
        </xsd:element>
      </xsd:choice>
    </xsd:complexContent>
  </xsd:extension>
</xsd:complexType>
```

Conformance Points 7

7.1 Summary of optional versus mandatory interfaces

All interfaces are mandatory within the compliance points. The interfaces are RepositoryManager, TargetManager, ExecutionManager, NodeManager, ApplicationManager, and Application.

7.2 Proposed conformance points

In general, the PIM suggests and enables several independent compliance points to enable different vendor implementations or user replacement of implementations. These are:

- RepositoryManager
  Rationale is that this function can be standalone, and implementations can offer a wide range of persistence, database, security, file system or web functionality.
- TargetManager
  Rationale is that this function can be standalone for independent offline planning or fully dynamic at runtime. Both could coexist.
- NodeManager
  Rationale is that this function is related to the node OS, ORB, development system etc., and there would likely be multiple vendors' implementations in a given distributed system. it should be a modest effort for a node platform supplier to implement this without the rest of the deployment system.
- ExecutionManager
  This is the core of the deployment system.

The PSMs define their own specific compliance points. For the CCM PSM, all 4 are defined.

In Chapter 2, the UML Profile for D&C Tool Support, suggests a further set of conformance points for tools:

- Modeling Tools that can create a well formed conformant M0 model of the PIM for CCM
- Forward Engineering Tools that can generate well formed XML, based on the XML schema for the PSM for CCM, of conformant M0 models.

7.3 Changes or extensions required to adopted OMG specifications

As intended, the CCM PSM replaces the “Packaging and Deployment” (and associated IDL) and “XML DTDs” chapters of CCM 3.0. The implications of this change are discussed in the migration subsection of the CCM PSM section.

7.4 Complete IDL definitions

Note that IDL definitions for the CCM PSM are generated based on the rules in the PSMs and are included in the appendix. The included IDL is normative due to the lack of tools to perform the mapping automatically.

References A

A.1 List of References

IDL for CCM B

B.1 Introduction

Chapter 5 describes the process to generate concrete IDL from the platform independent model, by using the rules defined by the UML Profile for CORBA on a transformation of the original PIM. With these rules, Chapter 5 contains the normative definition.

This chapter contains IDL that has been produced from the PIM using these rules. It is non-normative, so in the case of discrepancies, Chapter 5 is relevant.

B.2 Full IDL

The file orb.idl is used.

```
#include <orb.idl>
```

B.3 Exceptions

```
module Deployment {
 /*
  * Exceptions
  */

 exception PackageError {
   CORBA::StringSeq label;
   string reason;
 };

 exception ResourceNotAvailable {
   string label;
   string resourceType;
```

```
    string propertyName;
    string elementName;
    string resourceName;
  };

  exception PlanError {
    string label;
    string reaon;
  };

  exception StartError {
    string label;
    string reaon;
  };

  exception StopError {
    string label;
    string reaon;
  };

  exception InvalidProperty {
    string label;
    string reaon;
  };

  exception InvalidConnection {
    string label;
    string reaon;
  };

  exception NameExists {};
  exception NoSuchName {};
  exception LastConfiguration {};
  exception InvalidReference {};
};
```

B.4 Common Elements

```
module Deployment {
  struct Property {
    string name;
    any value;
  };

  typedef sequence<Property> Properties;

  struct Requirement {
    string resourceType;
    Properties property;
  };
  typedef sequence<Requirement> Requirements;

  enum SatisfierPropertyKind {
    Quantity,
    Capacity,
    Minimum,
    Maximum,
    Attribute,
    Selection
  };

  struct SatisfierProperty {
    string name;
    SatisfierPropertyKind kind;
    any value;
  };

  typedef sequence<SatisfierProperty> SatisfierProperties;

  struct RequirementSatisfier {
    string name;
    CORBA::StringSeq resourceType;
    SatisfierProperties property;
  };

  typedef sequence<RequirementSatisfier> RequirementSatisfiers;
};
```

B.5 Component Data Model

```
module Deployment {
  struct Capability {
    string name;
    CORBA::StringSeq resourceType;
    SatisfierProperties property;
  };

  typedef sequence<Capability> Capabilities;

  enum CCMComponentPortKind {
    Facet,
    SimplexReceptacle,
    MultiplexReceptacle,
    EventEmitter,
    EventPublisher,
    EventConsumer
  };

  struct ComponentPortDescription {
    string name;
    string specificType;
    CORBA::StringSeq supportedType;
    CCMComponentPortKind kind;
    boolean exclusiveProvider;
    boolean exclusiveUser;
    boolean optional;
  };

  typedef sequence<ComponentPortDescription> ComponentPortDescrip-
tions;

  struct ComponentPropertyDescription {
    string name;
    TypeCode type;
  };

  typedef sequence<ComponentPropertyDescription> ComponentProperty-
Descriptions;

  struct ComponentInterfaceDescription {
    string label;
    string UUID;
    string idlFile;
    string specificType;
    CORBA::StringSeq supportedType;
    ComponentPortDescriptions port;
    ComponentPropertyDescriptions property;
    Properties configProperty;
  };

  struct ImplementationArtifactDescription;
  typedef sequence<ImplementationArtifactDescription> ImplementationAr-
tifactDescriptions;

  struct ImplementationArtifactDescription {
    string label;
    string UUID;
    string location;
    ImplementationArtifactDescriptions dependsOn;
    Properties execParameter;
    Requirements deployRequirement;
  };

  struct MonolithicImplementationDescription {
    ImplementationArtifactDescriptions primaryArtifact;
    Properties execParameter;
    Requirements deployRequirement;
  };

  typedef sequence<MonolithicImplementationDescription> MonolithicIm-
```

```
plementationDescriptions;

  struct SubcomponentPropertyReference {
    string propertyName;
    unsigned long instanceRef;
  };

  typedef sequence<SubcomponentPropertyReference> Subcomponent-
PropertyReferences;

  struct AssemblyPropertyMapping {
    string label;
    string externalName;
    SubcomponentPropertyReferences delegatesTo;
  };

  typedef sequence<AssemblyPropertyMapping> AssemblyPropertyMap-
pings;

  struct ExternalReferenceEndpoint {
    string location;
  };

  typedef sequence<ExternalReferenceEndpoint> ExternalReferenceEnd-
points;

  struct SubcomponentPortEndpoint {
    string portName;
    unsigned long instanceRef;
  };

  typedef sequence<SubcomponentPortEndpoint> SubcomponentPortEnd-
points;

  struct ComponentExternalPortEndpoint {
    string portName;
  };

  typedef sequence<ComponentExternalPortEndpoint> ComponentExter-
nalPortEndpoints;

  struct AssemblyConnectionDescription {
    string label;
    ComponentExternalPortEndpoints externalEndpoint;
    SubcomponentPortEndpoints internalEndpoint;
    ExternalReferenceEndpoints externalReference;
    Requirements deployRequirement;
  };

  typedef sequence<AssemblyConnectionDescription> AssemblyConnec-
tionDescriptions;
```

```
  struct ComponentPackageReference {
    string requiredType;
  };

  typedef sequence<ComponentPackageReference> ComponentPack-
ageReferences;

  struct ComponentPackageDescription;
  typedef sequence<ComponentPackageDescription> ComponentPackage-
Descriptions;

  struct SubcomponentInstantiationDescription {
    string label;
    Properties configProperty;
    Requirements selectRequirement;
    ComponentPackageReferences reference;
    ComponentPackageDescriptions package;
  };

  typedef sequence<SubcomponentInstantiationDescription> Subcompo-
nentInstantiationDescriptions;

  struct ComponentAssemblyDescription {
    SubcomponentInstantiationDescriptions instance;
    AssemblyConnectionDescriptions connection;
    AssemblyPropertyMappings externalProperty;
  };

  typedef sequence<ComponentAssemblyDescription> ComponentAssem-
blyDescriptions;

  struct ComponentImplementationDescription {
    string label;
    string UUID;
    Properties configProperty;
    Capabilities capability;
    ComponentInterfaceDescription implements;
    ComponentAssemblyDescriptions assemblyImpl;
    MonolithicImplementationDescriptions monolithicImpl;
  };

  typedef sequence<ComponentImplementationDescription> ComponentIm-
plementationDescriptions;

  struct ComponentPackageDescription {
    string label;
    string UUID;
    Properties configProperty;
    ComponentInterfaceDescription realizes;
    ComponentImplementationDescriptions implementation;
  };

  struct PackageConfiguration;
  typedef sequence<PackageConfiguration> PackageConfigurations;

  struct PackageConfiguration {
    string label;
    string name;
    Properties configProperty;
    Requirements selectRequirement;
    PackageConfigurations specializedConfig;
    ComponentPackageDescriptions basePackage;
  };
};
```

B.6 Component Management Model

```
module Deployment {
  interface RepositoryManager {
    void installPackage (in string name,
                 in string label,
                 in string location)
      raises (NameExists, PackageError);

    PackageConfiguration findConfigurationByName (in string name)
      raises (NoSuchName);

    CORBA::StringSeq getAllNames ();
    CORBA::StringSeq findNamesByType (in string type);
    CORBA::StringSeq getAllTypes ();

    void createConfiguration (in string nname, in string bname,
                     in Properties cp, in Requirements sr)
      raises (NoSuchName, NameExists);

    void updateConfiguration (in string name,
                     in Properties cp, in Requirements sr)
      raises (NoSuchName);

    void deleteConfiguration (in string name, in boolean deletePackage)
      raises (NoSuchName, LastConfiguration);
  };
};
```

B.7 Target Data Model

```
module Deployment {
  struct Resource {
    string name;
    CORBA::StringSeq resourceType;
    SatisfierProperties property;
  };

  typedef sequence<Resource> Resources;

  struct SharedResource {
    string name;
    CORBA::StringSeq resourceType;
    SatisfierProperties property;
    CORBA::ULongSeq nodeRef;
  };

  typedef sequence<SharedResource> SharedResources;

  struct Node {
    string name;
    string label;
    Resources resource;
    CORBA::ULongSeq connectionRef;
    CORBA::ULongSeq sharedResourceRef;
  };

  typedef sequence<Node> Nodes;

  struct Interconnect {
    string name;
    string label;
    Resources resource;
    CORBA::ULongSeq connectionRef;
    CORBA::ULongSeq connectRef;
  };

  typedef sequence<Interconnect> Interconnects;

  struct Bridge {
    string name;
    string label;
    Resources resource;
    CORBA::ULongSeq connectRef;
  };

  typedef sequence<Bridge> Bridges;

  struct Domain {
    string UUID;
    string label;
    Nodes node;
    Interconnects interconnect;
    Bridges bridge;
    SharedResources sharedResource;
  };
};
```

B.8 Execution Data Model

```
module Deployment {
  struct PlanSubcomponentPropertyReference {
    string propertyName;
    unsigned long instanceRef;
  };

  typedef sequence<PlanSubcomponentPropertyReference> PlanSubcom-
ponentPropertyReferences;

  struct PlanPropertyMapping {
    string label;
    CORBA::StringSeq source;
    string externalName;
    PlanSubcomponentPropertyReferences delegatesTo;
  };

  typedef sequence<PlanPropertyMapping> PlanPropertyMappings;

  struct PlanSubcomponentPortEndpoint {
    string portName;
    CCMComponentPortKind kind;
    unsigned long instanceRef;
  };

  typedef sequence<PlanSubcomponentPortEndpoint> PlanSubcomponent-
PortEndpoints;

  struct PlanConnectionDescription {
    ComponentExternalPortEndpoints externalEndpoint;
    PlanSubcomponentPortEndpoints internalEndpoint;
    ExternalReferenceEndpoints externalReference;
  };

  typedef sequence<PlanConnectionDescription> PlanConnectionDescrip-
tions;

  struct ArtifactDeploymentDescription {
    string location;
    string label;
    string node;
    Properties execParameter;
    Requirements deployRequirement;
  };

  typedef sequence<ArtifactDeploymentDescription> ArtifactDeploymentDe-
scriptions;

  struct MonolithicDeploymentDescription {
    string label;
    CORBA::ULongSeq artifactRef;
    Requirements deployRequirement;
    Properties execParameter;
  };

  typedef sequence<MonolithicDeploymentDescription> MonolithicDeploy-
mentDescriptions;

  struct InstanceDeploymentDescription {
    string node;
    string label;
    unsigned long implementationRef;
    Properties configProperty;
  };

  typedef sequence<InstanceDeploymentDescription> InstanceDeployment-
Descriptions;

  struct DeploymentPlan {
    string label;
    ComponentInterfaceDescription realizes;
    ArtifactDeploymentDescriptions artifact;
    MonolithicDeploymentDescriptions implementation;
    InstanceDeploymentDescriptions instance;
    PlanConnectionDescriptions connection;
    PlanPropertyMappings externalProperty;
  };
};
```

B.9 Target Management Model

```
module Deployment {
  enum DomainUpdateKind {
    Add,
    Delete,
    UpdateAll,
    UpdateAvailable
  };

  interface TargetManager {
    Domain getAllResources ();
    Domain getAvailableResources ();

    void commitResources (in DeploymentPlan plan)
      raises (ResourceNotAvailable, PlanError);
```

```
void releaseResources (in DeploymentPlan plan)
  raises (PlanError);

void updateDomain (in CORBA::StringSeq elements,
               in Domain domainSupset,
               in DomainUpdateKind updateKind);
  };
};
```

B.10 Execution Management Model

```
module Deployment {
  interface Logger {};

  struct Connection {
    string name;
    CORBA::ObjectSeq endpoint;
  };

  typedef sequence<Connection> connections;

  interface Application {
    void finishLaunch (in Connections providedReference,
                in boolean start)
      raises (StartError);
    void start ();
  };

  typedef sequence<Application> Applications;

  interface DomainApplication : Application {};
  interface NodeApplication : Application {};

  interface ApplicationManager {
    Application startLaunch (in Properties configProperty,
                  out Connections providedReference)
      raises (InvalidProperty, StartError, ResourceNotAvailable);
    void destroyApplication (in Application app)
      raises (StopError, InvalidReference);
  };

  interface DomainApplicationManager : ApplicationManager {
    Applications getApplications ();
    DeploymentPlan getPlan ();
  };

  typedef sequence<DomainApplicationManager> DomainApplicationMan-
agers;

  interface NodeApplicationManager : ApplicationManager {
  };

  interface NodeManager {
    void joinDomain (in Domain domainSubset,
              in TargetManager manager,
              in Logger log);
    void leaveDomain ();

    NodeApplicationManager preparePlan (in DeploymentPlan plan)
      raises (StartError, PlanError);

    void destroyManager (in NodeApplicationManager manager)
      raises (StopError, InvalidReference);
  };

  interface ExecutionManager {
    DomainApplicationManager preparePlan (in DeploymentPlan plan)
      raises (StartError, PlanError, ResourceNotAvailable);

    void destroyManager (in DomainApplicationManager manager)
      raises (StopError, InvalidReference);

    DomainApplicationManagers getManagers ();
  };
};
```

XML Schema for CCM C

C.1 Introduction

Chapter 5, in combination with Chapter 6, describes the process to generate the concrete XML schema from the platform independent model. With these rules, chapters 5 and 6 contain normative definitions.

This chapter contains the XML schema that has been produced from the PIM using these rules. It is non-normative, so in the case of discrepancies, chapters 5 and 6 are relevant.

C.2 Full XML Schema

```
<?xml version="1.0" encoding="utf-8" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.omg.org/DnC"
            xmlns:DnC="http://www.omg.org/DnC">
```

C.3 Component Data Model

Any

```
  <xsd:complexType name="Any">
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:any processContents="skip" />
    </xsd:choice>
    <xsd:anyAttribute processContents="skip" />
</xsd:complexType>
```

Property

```
    <xsd:complexType name="Property">
      <xsd:sequence>
        <xsd:element name="name" type="xsd:string" />
        <xsd:element name="value" type="DnC:Any"
            minOccurs="1" maxOccurs="1" />
      </xsd:sequence>
  </xsd:complexType>
```

Requirement

```
    <xsd:complexType name="Requirement">
      <xsd:sequence>
        <xsd:element name="resourceType" type="xsd:string" />
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
          <xsd:element name="property" type="DnC:Property" />
        </xsd:choice>
      </xsd:sequence>
  </xsd:complexType>
```

PackageConfiguration

```
    <xsd:complexType name="PackageConfiguration">
      <xsd:sequence>
        <xsd:element name="name"  type="xsd:string" />
        <xsd:element name="label" type="xsd:string" />
        <xsd:choice minOccurs="0" maxOccurs="1">
          <xsd:element name="specializedConfig">
            <xsd:complexType>
              <xsd:attribute name="href" type="xsd:string"
                             use="required">
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="basePackage">
            <xsd:complexType>
              <xsd:attribute name="href" type="xsd:string"
                             use="required">
            </xsd:complexType>
          </xsd:element>
        </xsd:choice>
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
          <xsd:element name="selectRequirement" type="DnC:Requirement" />
        </xsd:choice>
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
          <xsd:element name="configProperty" type="DnC:Property" />
        </xsd:choice>
      </xsd:sequence>
  </xsd:complexType>
```

ComponentPackageDescription

```
<xsd:complexType name="ComponentPackageDescription">
  <xsd:sequence>
    <xsd:element name="label" type="xsd:string" />
    <xsd:element name="uuid"  type="xsd:string" />
    <xsd:element name="realizes">
      <xsd:complexType>
        <!-- ComponentInterfaceDescription pointer -->
        <xsd:attribute name="href" type="xsd:string"
                       use="required">
      </xsd:complexType>
    </xsd:element>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="implementations">
        <xsd:complexType>
          <!-- ComponentImplementationDescription pointer -->
          <xsd:attribute name="href" type="xsd:string"
                         use="required">
        </xsd:complexType>
      </xsd:element>
    </xsd:choice>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="configProperty" type="DnC:Property" />
    </xsd:choice>
  </xsd:sequence>
```

```
</xsd:complexType>
```

ComponentImplementationDescription

```
<xsd:complexType name="ComponentImplementationDescription">
  <xsd:sequence>
    <xsd:element name="label" type="xsd:string" />
    <xsd:element name="uuid"  type="xsd:string" />
    <xsd:choice minOccurs="0" maxOccurs="1">
      <xsd:element name="assemblyImpl">
        <xsd:complexType>
          <!-- ComponentAssemblyDescription pointer -->
          <xsd:attribute name="href" type="xsd:string"
                         use="required">
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="monolithicImpl">
        <xsd:complexType>
          <!-- MonolithicImplementationDescription pointer -->
          <xsd:attribute name="href" type="xsd:string"
                         use="required">
        </xsd:complexType>
      </xsd:element>
    </xsd:choice>
    <xsd:element name="implements">
      <xsd:complexType>
        <!-- ComponentInterfaceDescription pointer -->
        <xsd:attribute name="href" type="xsd:string"
                       use="required">
      </xsd:complexType>
    </xsd:element>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="configProperty" type="DnC:Property" />
    </xsd:choice>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="capability" type="DnC:Capability" />
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

ComponentAssemblyDescription

```
<xsd:complexType name="ComponentAssemblyDescription">
  <xsd:sequence>
    <xsd:choice minOccurs="1" maxOccurs="unbounded">
      <xsd:element name="instance"
                   type="DnC:SubcomponentInstantiationDescription" />
    </xsd:choice>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="connection"
                   type="DnC:AssemblyConnectionDescription" />
    </xsd:choice>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="externalProperty"
                   type="DnC:AssemblyPropertyMapping" />
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

SubcomponentInstantiationDescription

```
<xsd:complexType name="SubcomponentInstantiationDescription">
  <xsd:sequence>
    <xsd:choice minOccurs="0" maxOccurs="1">
      <xsd:element name="reference">
        <xsd:complexType>
          <!-- ComponentPackageReference pointer -->
          <xsd:attribute name="href" type="xsd:string"
                         use="required">
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="package">
        <xsd:complexType>
          <!-- ComponentPackageDescription pointer -->
          <xsd:attribute name="href" type="xsd:string"
```

```
                                  use="required">
              </xsd:complexType>
            </xsd:element>
          </xsd:choice>
          <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="configProperty" type="DnC:Property" />
          </xsd:choice>
          <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="selectRequirement" type="DnC:Requirement" />
          </xsd:choice>
        </xsd:sequence>
       </xsd:complexType>
```

ComponentPackageReference

```
  <xsd:complexType name="ComponentPackageReference">
    <xsd:choice minOccurs="1" maxOccurs="1">
      <xsd:element name="requiredType" type="xsd:string" />
    </xsd:choice>
</xsd:complexType>
```

AssemblyConnectionDescription

```
<xsd:complexType name="AssemblyConnectionDescription">
  <xsd:sequence>
    <xsd:element name="label" type="xsd:string" />
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="externalEndpoint"
                   type="DnC:ComponentExternalEndpoint" />
    </xsd:choice>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="internalEndpoint"
                   type="DnC:SubcomponentPortEndpoint" />
    </xsd:choice>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="externalReference"
                   type="DnC:ExternalReferenceEndpoint" />
    </xsd:choice>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="deployRequirement" type="DnC:Requirement" />
  </xsd:sequence>
</xsd:complexType>
```

ComponentExternalEndpoint

```
<xsd:complexType name="ComponentExternalEndpoint">
  <xsd:choice minOccurs="1" maxOccurs="1">
    <xsd:element name="portName"  type="xsd:string" />
  </xsd:choice>
</xsd:complexType>
```

SubcomponentPortEndpoint

```
<xsd:complexType name="SubcomponentPortEndpoint">
  <xsd:sequence>
    <xsd:element name="portName"  type="xsd:string" />
    <xsd:element name="instance">
      <xsd:complexType>
        <!-- ComponentAssemblyDescription pointer -->
        <xsd:attribute name="href" type="xsd:string"
                       use="required">
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

ExternalReferenceEndpoint

```
<xsd:complexType name="ExternalReferenceEndpoint">
  <xsd:choice minOccurs="1" maxOccurs="1">
    <xsd:element name="location"  type="xsd:URL" />
  </xsd:choice>
</xsd:complexType>
```

AssemblyPropertyMapping

```
<xsd:complexType name="AssemblyPropertyMapping">
  <xsd:sequence>
```

```
        <xsd:element name="label" type="xsd:string" />
        <xsd:element name="externalName" type="xsd:string" />
        <xsd:choice minOccurs="1" maxOccurs="unbounded">
          <xsd:element name="delegatedProperty">
            <xsd:complexType>
              <xsd:element name="name" type="xsd:string" />
              <xsd:element name="instance">
                <xsd:complexType>
                  <!-- Ref. to SubcomponentInstantiationDescription -->
                  <xsd:attribute name="href" type="xsd:string"
                                 use="required" />
                </xsd:complexType>
              </xsd:element>
            </xsd:complexType>
          </xsd:element>
        </xsd:choice>
      </xsd:sequence>
    </xsd:complexType>
```

MonolithicImplementationDescription

```
    <xsd:complexType name="MonolithicImplementationDescription">
      <xsd:sequence minOccurs="1" maxOccurs="1">
        <xsd:choice minOccurs="1" maxOccurs="unbounded">
          <xsd:element name="primaryArtifact">
            <xsd:complexType>
              <!-- ImplementationArtifactDescription pointer -->
              <xsd:attribute name="href" type="xsd:string"
                             use="required">
            </xsd:complexType>
          </xsd:element>
        </xsd:choice>
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
          <xsd:element name="execParameter" type="DnC:Property" />
        </xsd:choice>
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
          <xsd:element name="deployRequirement" type="DnC:Requirement" />
        </xsd:choice>
      </xsd:sequence>
    </xsd:complexType>
```

ImplementationArtifactDescription

```
<xsd:complexType name="ImplementationArtifactDescription">
  <xsd:sequence>
    <xsd:element name="label" type="xsd:string" />
    <xsd:element name="uuid" type="xsd:string" />
    <xsd:element name="location" type="xsd:URL" />
    <xsd:element name="describes">
      <xsd:complexType>
        <!-- ImplementationArtifact pointer -->
        <xsd:attribute name="href" type="xsd:string"
                       use="required">
      </xsd:complexType>
    </xsd:element>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="dependsOn">
        <xsd:complexType>
          <!-- ImplementationArtifactDescription pointer -->
          <xsd:attribute name="href" type="xsd:string"
                         use="required">
        </xsd:complexType>
      </xsd:element>
    </xsd:choice>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="execParameter" type="DnC:Property" />
    </xsd:choice>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="deployRequirement" type="DnC:Requirement" />
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

Implementation Artifact

```
<xsd:complexType name="ImplementationArtifact">
  <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:element name="dependsOn">
      <xsd:complexType>
        <!-- ImplementationArtifactDescription pointer -->
        <xsd:attribute name="href" type="xsd:string"
                       use="required">
      </xsd:complexType>
    </xsd:element>
  </xsd:choice>
</xsd:complexType>
```

ComponentInterfaceDescription

```
<xsd:complexType name="ComponentInterfaceDescription">
  <xsd:sequence>
    <xsd:element name="label" type="xsd:string" />
    <xsd:element name="uuid"  type="xsd:string" />
    <xsd:element name="specificType"  type="xsd:string" />
    <xsd:choice minOccurs="1" maxOccurs="unbounded">
      <xsd:element name="supportedType"  type="xsd:string" />
    </xsd:choice>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="port" type="DnC:ComponentPortDescription" />
    </xsd:choice>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="property" type="DnC:Property" />
    </xsd:choice>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="configProperty" type="DnC:Property" />
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

ComponentPortDescription

```
<xsd:complexType name="ComponentPortDescription">
```

```
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string" />
    <xsd:element name="specificType"  type="xsd:string" />
    <xsd:choice minOccurs="1" maxOccurs="unbounded">
      <xsd:element name="supportedType"  type="xsd:string" />
    </xsd:choice>
    <xsd:element name="provider" type="boolean" />
    <xsd:element name="exclusiveProvider" type="boolean" />
    <xsd:element name="exclusiveUser" type="boolean" />
    <xsd:element name="optional" type="xsd:string" />
   </xsd:all>
   <xsd:attribute name="id" type="xsd:string" />
</xsd:complexType>
```

ComponentPropertyDescription

```
<xsd:complexType name="ComponentPropertyDescription">
  <xsd:sequence>
    <xsd:element name="name"  type="xsd:string" />
    <xsd:element name="type" type="xsd:string" />
  </xsd:sequence>
</xsd:complexType>
```

RequirementSatisfier

```
<xsd:complexType name="RequirementSatifier">
  <xsd:sequence>
    <xsd:element name="name"  type="xsd:string" />
    <xsd:choice minOccurs="1" maxOccurs="unbounded">
      <xsd:element name="resourceType"  type="xsd:string" />
    </xsd:choice>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="property" type="DnC:SatisfierProperty" />
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

Capability

```
<xsd:complexType name="Capability">
  <xsd:extension base="DnC:RequirementSatisfier">
  </xsd:extension>
</xsd:complexType>
```

SatisfierProperty

```
<xsd:complexType name="SatifierProperty">
  <xsd:sequence>
    <xsd:element name="name"  type="xsd:string" />
    <xsd:element name="value"  type="DnC:Any" />
  </xsd:sequence>
  <xsd:attribute name="kind" type="DnC:SatisfierPropertyKind"
                 use="required" />
</xsd:complexType>
<xsd:simpleType name="SatisfierPropertyKind">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="quantity" />
    <xsd:enumeration value="capacity" />
    <xsd:enumeration value="minimum" />
    <xsd:enumeration value="maximum" />
    <xsd:enumeration value="attribute" />
    <xsd:enumeration value="selection" />
  </xsd:restriction>
</xsd:simpleType>
```

C.4 Target Data Model

Domain

```
<xsd:element name="domain" type="DnC:Domain" minOccurs="0" />
<xsd:complexType name="Domain">
  <xsd:sequence>
    <xsd:element name="uuid" type="xsd:string" />
    <xsd:element name="label" type="xsd:string" />
    <xsd:element name="nodes">
      <xsd:complexType>
```

```
          <xsd:choice minOccurs="1" maxOccurs="unbounded">
            <xsd:element name="node" type="DnC:Node">
          </xsd:choice>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="interconnects">
        <xsd:complexType>
          <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="interconnect" type="DnC:Interconnect">
          </xsd:choice>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="bridges">
        <xsd:complexType>
          <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="bridge" type="DnC:Bridge">
          </xsd:choice>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="sharedResources">
        <xsd:complexType>
          <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="sharedResource"
                                type="DnC:SharedResource">
          </xsd:choice>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </complexType>
```

Resource

```
  <xsd:complexType name="Resource">
    <xsd:extension base="DnC:RequirementSatisfier">
    </xsd:extension>
  </xsd:complexType>
```

SharedResource

```
<xsd:complexType name="SharedResource">
  <xsd:extension base="DnC:Resource">
    <xsd:complexContent>
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="node">
          <xsd:complexType>
            <!-- Hosting Node pointer -->
            <xsd:attribute name="href" type="xsd:string"
                           use="required">
          </xsd:complexType>
        </xsd:element>
      </xsd:choice>
    </xsd:complexContent>
  </xsd:extension>
</xsd:complexType>
```

Node

```
<xsd:complexType name="Node">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string" />
    <xsd:element name="label" type="xsd:string" />
    <xsd:choice minOccurs="0" maxOccurs="unbounded" />
      <xsd:element name="resource" type="DnC:Resource" />
    </xsd:choice>
    <xsd:choice minOccurs="0" maxOccurs="unbounded" />
      <xsd:element name="sharedResource">
        <xsd:complexType>
          <!-- SharedResource pointer -->
          <xsd:attribute name="href" type="xsd:string"
                         use="required">
        </xsd:complexType>
      </xsd:element>
    </xsd:choice>
    <xsd:choice minOccurs="0" maxOccurs="unbounded" />
      <xsd:element name="connection">
```

```
          <xsd:complexType>
            <!-- Interconnect pointer -->
            <xsd:attribute name="href" type="xsd:string"
                           use="required">
          </xsd:complexType>
        </xsd:element>
      </xsd:choice>
    </xsd:sequence>
</xsd:complexType>
```

Interconnect

```
<xsd:complexType name="Interconnect">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string" />
    <xsd:element name="label" type="xsd:string" />
    <xsd:choice minOccurs="0" maxOccurs="unbounded" />
      <xsd:element name="resource" type="DnC:Resource" />
    </xsd:choice>
    <xsd:choice minOccurs="1" maxOccurs="unbounded" />
      <xsd:element name="node">
        <xsd:complexType>
          <!-- Connection to Node pointer -->
          <xsd:attribute name="href" type="xsd:string"
                         use="required">
        </xsd:complexType>
      </xsd:element>
    </xsd:choice>
    <xsd:choice minOccurs="0" maxOccurs="unbounded" />
      <xsd:element name="bridge">
        <xsd:complexType>
          <!-- Connection to Bridge pointer -->
          <xsd:attribute name="href" type="xsd:string"
                         use="required">
        </xsd:complexType>
      </xsd:element>
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

Bridge

```
<xsd:complexType name="Bridge">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string" />
    <xsd:element name="label" type="xsd:string" />
    <xsd:choice minOccurs="0" maxOccurs="unbounded" />
      <xsd:element name="resource" type="DnC:Resource" />
    </xsd:choice>
    <xsd:choice minOccurs="1" maxOccurs="unbounded" />
      <xsd:element name="connect">
        <xsd:complexType>
          <!-- Connection to Interconnect pointer -->
          <xsd:attribute name="href" type="xsd:string"
                         use="required">
        </xsd:complexType>
      </xsd:element>
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

C.5 Execution Data Model

DeploymentPlan

```
<xsd:element name="deploymentPlan" type="DnC:DeploymentPlan"
             minOccurs="0" />
  <xsd:complexType name="DeploymentPlan">
    <xsd:sequence>
      <xsd:element name="label" type="xsd:string" />
      <!-- XLink to ComponentInterfaceDescription -->
      <xsd:element name="realizes">
        <xsd:complexType>
          <xsd:attribute name="href" type="xsd:string"
                         use="required">
        </xsd:complexType>
      </xsd:element>
      <!-- List of Artifacts -->
      <xsd:element name="artifacts">
        <xsd:complexType>
          <xsd:choice minOccurs="1" maxOccurs="unbounded">
            <xsd:element name="artifact">
              <xsd:complexType>
                <xsd:complexContent>
                  <xsd:extension
                       base="DnC:ArtifactDeploymentDescription">
                    <attribute name="ïd" type="xsd:string"
                               use="required" />
                  </xsd:extension>
                </xsd:complexContent>
              </complexType>
            </xsd:element>
          </xsd:choice>
        </xsd:complexType>
      </xsd:element>
      <!-- List of Implementations -->
      <xsd:element name="implementations">
        <xsd:complexType>
          <xsd:choice minOccurs="1" maxOccurs="unbounded">
            <xsd:element name="implementation">
              <xsd:complexType>
                <xsd:complexContent>
                  <xsd:extension
                       base="DnC:MonolithicDeploymentDescription">
                    <attribute name="ïd" type="xsd:string"
                               use="required" />
                  </xsd:extension>
                </xsd:complexContent>
              </complexType>
            </xsd:element>
          </xsd:choice>
        </xsd:complexType>
      </xsd:element>
      <!-- List of Instances -->
      <xsd:element name="instances">
        <xsd:complexType>
          <xsd:choice minOccurs="1" maxOccurs="unbounded">
            <xsd:element name="instance">
              <xsd:complexType>
                <xsd:complexContent>
                  <xsd:extension
                       base="DnC:InstanceDeploymentDescription">
                    <attribute name="ïd" type="xsd:string"
                               use="required" />
                  </xsd:extension>
                </xsd:complexContent>
              </complexType>
            </xsd:element>
          </xsd:choice>
        </xsd:complexType>
      </xsd:element>
      <!-- List of Connections -->
      <xsd:element name="connections">
        <xsd:complexType>
          <xsd:choice minOccurs="1" maxOccurs="unbounded">
            <xsd:element name="connection">
              <xsd:complexType>
                <xsd:complexContent>
                  <xsd:extension base="DnC:PlanConnectionDescription">
                    <attribute name="ïd" type="xsd:string"
                               use="required" />
                  </xsd:extension>
                </xsd:complexContent>
              </complexType>
            </xsd:element>
          </xsd:choice>
        </xsd:complexType>
```

```
      </xsd:element>
      <!-- List of Property Mappings -->
      <xsd:element name="propertyMappings">
        <xsd:complexType>
          <xsd:choice minOccurs="1" maxOccurs="unbounded">
            <xsd:element name="externalProperty">
              <xsd:complexType>
                <xsd:complexContent>
                  <xsd:extension base="DnC:PlanPropertyMapping">
                    <attribute name="ïd" type="xsd:string"
                               use="required" />
                  </xsd:extension>
                </xsd:complexContent>
              </complexType>
            </xsd:element>
          </xsd:choice>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
```

ArtifactDeploymentDescription

```
  <xsd:complexType name="ArtifactDeploymentDescription">
    <xsd:sequence>
      <xsd:element name="location" type="xsd:URL" />
      <xsd:element name="label" type="xsd:string" />
      <xsd:element name="node" type="xsd:string" />
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="deployRequirement" type="DnC:Requirement" />
      </xsd:choice>
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="execParameter" type="DnC:Property" />
      </xsd:choice>
    </xsd:sequence>
</xsd:complexType>
```

MonolithicDeploymentDescription

```
<xsd:complexType name="MonolithicDeploymentDescription">
  <xsd:sequence>
    <xsd:element name="label" type="xsd:string" />
    <xsd:sequence minOccurs="1" maxOccurs="unbounded">
      <xsd:element name="implementingArtifact">
        <!-- Reference to Artifact -->
        <attribute name="idref" type="xsd:IDREF" use="required" />
      </xsd:element>
    </xsd:sequence>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="deployRequirement" type="DnC:Requirement" />
    </xsd:choice>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="execParameter" type="DnC:Property" />
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

InstanceDeploymentDescription

```
<xsd:complexType name="InstanceDeploymentDescription">
  <xsd:sequence>
    <xsd:element name="node" type="xsd:string" />
    <xsd:element name="label" type="xsd:string" />
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="deployRequirement" type="DnC:Requirement" />
    </xsd:choice>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="execParameter" type="DnC:Property" />
    </xsd:choice>
  </xsd:sequence>
  <!-- Reference to MonolithicDeploymentDescription -->
  <attribute name="implementation" type="xsd:IDREF" use="required" />
</xsd:complexType>
```

PlanConnectionDescription

```
<xsd:complexType name="PlanConnectionDescription">
  <xsd:sequence>
    <xsd:element name="label" type="xsd:string" />
    <!-- Sequence of AssemblyConnectionDescription labels -->
    <xsd:element name="sourceList">
      <xsd:complexType>
        <xsd:sequence minOccurs="1" maxOccurs="unbounded">
          <xsd:element name="source" type="xsd:string" />
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="deployRequirement" type="DnC:Requirement" />
    </xsd:choice>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:choice>
      <xsd:element name="externalEndpoint"
                   type="DnC:ComponentExternalPortEndpoint" />
      <xsd:element name="internalEndpoint"
                   type="DnC:PlanSubcomponentPortEndpoint" />
      <xsd:element name="externalReference"
                   type="DnC:ExternalReferenceEndpoint" />
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

PlanSubcomponentPortEndpoint

```
<xsd:complexType name="PlanSubcomponentPortEndpoint">
  <xsd:sequence>
    <xsd:element name="portName" type="xsd:string" />
    <xsd:element name="provider" type="xsd:boolean" />
    <xsd:element name="instance">
      <xsd:complexType>
        <!-- Reference to the InstanceDeploymentDescription -->
        <xsd:attribute name="idref" type="xsd:IDREF"
```

```
                                      use="required" />
                  </xsd:complexType>
                </xsd:element>
              </xsd:sequence>
            </xsd:complexType>
```

PlanPropertyMapping

```
            <xsd:complexType name="PlanPropertyMapping">
              <xsd:sequence>
                <xsd:element name="label" type="xsd:string" />
                <!-- Sequence of AssemblyPropertyMapping labels -->
                <xsd:element name="sourceList">
                  <xsd:complexType>
                    <xsd:sequence minOccurs="1" maxOccurs="unbounded">
                      <xsd:element name="source" type="xsd:string" />
                    </xsd:sequence>
                  </xsd:complexType>
                </xsd:element>
                <xsd:element name="externalName" type="xsd:string" />
                <xsd:choice minOccurs="1" maxOccurs="unbounded">
                  <xsd:element name="delegatedProperty">
                    <xsd:complexType>
                      <xsd:element name="name" type="xsd:string" />
                      <xsd:element name="instance">
                        <xsd:complexType>
                          <!-- Reference to the InstanceDeploymentDescription -->
                          <xsd:attribute name="idref" type="xsd:IDREF"
                                         use="required" />
                        </xsd:complexType>
                      </xsd:element>
                    </xsd:complexType>
                  </xsd:element>
                </xsd:choice>
              </xsd:sequence>
            </xsd:complexType>
          </xsd:schema>
```