

Specification for Deployment and Configuration of Component-based Distributed Applications

Proposal to the OMG MARS RFP: Deployment and Configuration of Component-based Distributed Applications

Joint Revised Submission

Submitters

- Fraunhofer FOKUS
- Mercury Computer Systems
- Rockwell Collins

Supporters

- Raytheon Company
- MITRE Corporation
- BAE Systems
- ITT Industries
- 88solutions Corporation
- Deutsche Telekom
- France Telekom
- Humboldt Universität Berlin
- Carleton University
- Laboratoire d'Informatique Fondamentale de Lille

OMG Document mars/2003-03-xx, March 3, 2003

Copyright 2002-2003, Mercury Computer Systems, Inc.

Copyright 2002-2003, Rockwell Collins

Copyright 2002-2003, Fraunhofer FOKUS

The companies listed above hereby grant a royalty-free license to the Object Management Group, Inc. (OMG) for worldwide distribution of this document or any derivative works thereof, so long as the OMG reproduces the copyright notices and the below paragraphs on all distributed copies. The material in this document is submitted to the OMG for evaluation. Submission of this document does not represent a commitment to implement any portion of this specification in the products of the submitters.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND

WITH REGARD TO THIS MATERIAL INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

The companies listed above shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material. The information contained in this document is subject to change without notice.

This document contains information which is protected by copyright. All Rights Reserved. Except as otherwise provided herein, no part of this work may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without the permission of one of the copyright owners.

All copies of this document must include the copyright and other information contained on this page.

The copyright owners grant member companies of the OMG permission to make a limited number of copies of this document (up to fifty copies) for their internal use as part of the OMG evaluation process.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Right in Technical Data and Computer Software Clause at DFARS 252.227.7013.

CORBA, OMG, and Object Request Broker are trademarks of Object Management Group.

1. Contents	iii
2. Preface	vi
2.1 Copyright Waiver	vi
2.2 Co-submitting Companies and Supporters	vi
2.3 Submission Contact Points	vii
2.4 Submission Overview	vii
2.5 Design Rationale	viii
2.6 Statement of proof of concept.	viii
2.7 Resolution of RFP Mandatory and Optional requirements.	viii
2.8 Responses to RFP issues to be discussed	x
3. Introduction	1
3.1 Component-based Applications	1
3.2 The Target Environment	2
3.3 The Deployment Process	3
4. Platform Independent Model	6
4.1 Segmentation of the Model	6
4.2 Model Diagram Conventions	9
4.3 Component Data Model	9
4.4 Component Management Model.	24
4.5 Target Data Model	26
4.6 Target Management Model.	33
4.7 Execution Data Model	34
4.8 Execution Management Model.	37
4.9 Relations to Other Standards	38

5. Actors	40
5.1 Development Actors Overview	40
5.2 Designer	41
5.3 Implementor	41
5.4 Assembler	42
5.5 Packager	42
5.6 Domain Administrator	43
5.7 Deployment Actors Overview	44
5.8 Repository Administrator	44
5.9 Planner	45
5.10 Executor	48
6. PSM for SCA	49
6.1 About the SCA	49
6.2 Introduction.	49
6.3 Meta-Concepts	49
6.4 Use Case Mapping	50
6.5 Mapping Ideas.	50
7. PSM for CCM	52
7.1 Introduction.	52
7.2 Meta-Concepts	52
7.3 Component Data Model	53
7.4 Component Management Model	55
7.5 Target Data Model	55
7.6 Target Management Model.	56
7.7 Execution Data Model	56
7.8 Execution Management Model.	56
7.9 Miscellaneous	57
7.10 Impact on the CCM Specification	58
8. Mapping to XML Schema	59
9. Metamodel and MDA	60
9.1 Overview.	60
9.2 D&C Metamodel.	60
9.3 Definition of metaclasses	61
9.4 Relationship between metaclasses and PIM classes.	61
10. Use Cases	64
11. Conformance Points	72

11.1 Summary of optional versus mandatory interfaces	72
11.2 Proposed compliance points	72
11.3 Changes or extensions required to adopted OMG specifications	72
11.4 Complete IDL definitions	72
12. References	73

2.1 Copyright Waiver

The companies listed above hereby grant a royalty-free license to the Object Management Group, Inc. (OMG) for worldwide distribution of this document or any derivative works thereof, so long as the OMG reproduces the copyright notices and the below paragraphs on all distributed copies. The material in this document is submitted to the OMG for evaluation. Submission of this document does not represent a commitment to implement any portion of this specification in the products of the submitters.

The copyright owners grant member companies of the OMG permission to make a limited number of copies of this document (up to fifty copies) for their internal use as part of the OMG evaluation process.

2.2 Co-submitting Companies and Supporters

The following companies are pleased to co-submit the Specification for Deployment and Configuration of Component-based Distributed Applications (hereafter referred to as D&C Specification) in response to the MARS Task Force RFP - Deployment and Configuration of Component-based Distributed Applications:

- Fraunhofer FOKUS
- Mercury Computer Systems
- Rockwell Collins

The following companies are pleased to support the Specification for Deployment and Configuration of Component-based Distributed Applications in response to the MARS Task Force RFP - Deployment and Configuration of Component-based Distributed Applications:

- Raytheon Company
- MITRE Corporation
- BAE Systems
- ITT Industries
- 88solutions Corporation
- Deutsche Telekom
- France Telekom
- Humboldt Universität Berlin
- Carleton University
- Laboratoire d'Informatique Fondamentale de Lille

2.3 Submission Contact Points

Please send comments on this submission to this e-mail: deployment@omg.org.

2.4 Submission 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 into 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 submission defines a Platform Independent Model (PIM) to address the above issues, which introduces a conceptual basis for deployment systems independent of technology platform. This submission also defines Platform Specific Models for deployment and configuration for both the CORBA Component Model (CCM) and the Software Communication Architecture (SCA).

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 does not further define interfaces between elements of the infrastructure and thus does not necessarily enable interoperable implementations of parts of a deployment infrastructure.

Section 9 of this document describes the metamodel for the PIM. Section 4 describes the PIM. Section 7 describes the PSM for the CCM. Section 6 describes the PSM for the SCA. Section 11 discusses compliance issues. Section 8 describes the approach to creating XML schemas from models.

2.5 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.

We also limit the scope of this specification to interfaces to the distributed infrastructure as a whole, rather than define infrastructure interfaces internal to the distributed infrastructure. This is due to the wide range of implementation approaches and performance trade-offs that are possible and useful. Thus the specification does not necessarily provide interoperability between infrastructures on individual computers that comprise the distributed environment. If additional interoperability in this area is strongly desired by reviewers of this revised submission, a “default” set of interfaces could be defined which could provide for a “least common denominator” interoperability so that individual platforms could have deployment infrastructure elements developed by different vendors.

2.6 Statement of proof of concept

Submitters have experience with implementations of component deployment systems that are precursors to this specification. No aspect of this specification requires significant functionality not represented in those previous efforts. Previous efforts and experience that contributed to this specification include implementations of CCM, implementations of the SCA, and implementations of proprietary systems for high performance embedded systems (SCE — Streaming Component Environment).

2.7 Resolution of RFP Mandatory and Optional requirements

Mandatory Requirements

The list below describes how the submission meets the mandatory requirements, as put forth in Section 6.5, Mandatory Requirements, of the RFP

Requirement 1: Proposals shall define a terminology to describe the software and hardware infrastructures of heterogeneous distributed execution environments in terms of metamodel(s). The metamodel(s) shall at least provide elements for the description of computing nodes, network connections, available computing services on computing nodes, properties of computing nodes and network connections between computing nodes.

How the submission meets this requirement: This submission defines a “target” model that includes definitions of nodes and interconnections among nodes. Both are defined as having “resources” which encompass both properties and services relevant to deployment.

Requirement 2: Proposals shall define platform independent model(s) that support several potential configurations, required properties and features as well as distribution constraints for the deployment of components and assemblies of components onto distributed execution infrastructures.

How the submission meets this requirement: This submission defines alternative, hierarchical, implementations of components (assemblies), as well as multiple possible configurations of packages of component software. Deployment requirements are associated with a number of different entities in the packaged software. Feature tags can be used to associate implementations with features that can be requested by deployment tools.

Requirement 3: Proposals shall provide platform independent model(s) which describe the application and infrastructure facilities (e.g. interfaces, supportive services, etc.) necessary to automate the entire deployment process of component-based distributed applications and their life-cycle and configuration management.

How the submission meets this requirement: Interfaces and associated semantics are defined for installing, configuring, planning, preparing, launching, monitoring and terminating applications. These interfaces support different levels of automation ranging from offline and static deployment planning, to online, dynamic automation. Configuration management is supported by allowing multiple packages to coexist that implement the same application, and to have multiple configurations for each package.

Requirement 4: Proposals shall provide platform specific models for CCM by specializing the platform independent models of requirements 2 and 3 for CORBA components.

How the submission meets this requirement: Section 7 provides a PSM for CCM.

Requirement 5: In particular, the platform independent and platform specific models shall define facilities to:

- Automatically deploy assemblies of components onto the targeted distributed execution infrastructure.
- Automatically install and uninstall assemblies of components.
- Execute, as well as manage and control, an assembly of components during its run-time, including creation and destruction of instances of components contained by an assembly, as well as establishment and tear-down of connections between the components contained by an assembly,
- establish the initial configuration of components after their creation,
- initiate execution and termination of components on computing nodes of the distributed execution infrastructure,
- ensure that resources can be recovered when a deployment activity cannot be completed.

How the submission meets this requirement: The models defined support these activities, although the internal mechanisms used are not specified.

Requirement 6: Proposals shall specify facilities for information retrieval based on requirement 1) in order to support automated deployment of components or assemblies onto the targeted infrastructure. In particular, the models submitted against requirements 3 and 4 shall specify facilities to dynamically obtain information on available computing nodes within a distributed execution infrastructure, and the properties, resources and available computing services of these nodes, installed and running assemblies, and their actual configuration and topology.

How the submission meets this requirement: Interfaces are defined to obtain information about the target environment, including nodes, interconnections and the resources available from both. The run-time infrastructure provides interfaces to discover the applications that are installed, those that are deployed, and running, and how the running applications were deployed.

Optional Requirements

Optional Requirement 1: Proposals may provide Platform specific models for other technologies than CORBA by specializing the platform independent models of section 6.5. The platform specific models must define the semantics of a component as well as the semantics of assemblies of components in the context of automated deployment if there are differences to the definitions provided by CCM.

How the submission meets this requirement: This document defines a PSM for the SCA (Software Communication Architecture) in section 6.

Optional Requirement 2: 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.

How the submission meets this requirement: This specification defines XML schemas for the information describing packaged software (predeployment), the target environment, as well as the specification of exactly how the application software is intended to be deployed. In all cases the XML schema is derived from the model by a rule-based transformation.

Optional Requirement 3: Proposals may address life-cycle and dynamic reconfiguration management aspects during an application's run-time by providing appropriate Platform independent and Platform specific models.

How the submission meets this requirement: This specification defined models for starting, monitoring, and terminating deployed applications. Dynamic reconfiguration is *not* addressed in this submission.

Optional Requirement 4: Proposals may support the deployment and configuration of heterogeneous assemblies made up of components for different specific platforms (e.g. CORBA and non-CORBA environments).

How the submission meets this requirement: This specification does not address this optional requirement, although the PIM could be used as a basis for a PSM that could support this level of generality.

2.8 Responses to RFP issues to be discussed

Issue to be discussed from RFP: Proposals should point out what approaches are available for profiling or creating minimized versions of the submitted specification.

Submitters of this proposal are also submitters on the Lightweight CCM RFP, and they believe that this specification should not require profiling to meet the requirements of that RFP.

"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 subcomponents must first be created, then interconnected and configured. This specification deals with the deployment and configuration of component-based applications into 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 independent. 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.

3.1 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 removes the necessity 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 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 where they will be executed on, e.g. the CPU type, or on 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.

3.2 The Target Environment

The target environment is described as 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 Raceway bus. Bridges route between interconnects, representing both routers and switches.

Nodes, interconnects and bridges have resources that define their features, capabilities 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.

Information about the target environment can be both static or online. The static data describes total available resources and capacities in the domain, whereas online data reflects current resource usage. Static data is useful for planning ahead in a predictable environment (e.g. in an embedded system that only executes one application at a time), while online data is required in a dynamic distributed system where resources are used and released continuously.

3.3 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*.

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.

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.

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.

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 execution since all decisions can be made in advance (when 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 also, 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.

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 according to 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.

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. In this executing state, the application runs until it completes or is terminated via the same infrastructure that launched it.

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.

4.1 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.

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 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 argument 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.

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 process defined above. This should allow the different segments to be isolated according to usage (“need to know”), 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 breaks down nicely in terms of partitioning 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 (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 on 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 networks.

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.

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 section.

Table 1: D&C Model Segmentation Summary

	Data Model	Management/Runtime Model	Deployment Process Usage
	Can, in PSMs, imply/generate XML Schemas and IDL data definitions	Can imply interface IDL that may use data IDL derived from Data model. “Manager” applied to class names for consistency.	How/when are the models used in the deployment process. “Tool” is used here for the client that performs and controls the process.

Table 1: D&C Model Segmentation Summary

Component Software	Component Data Model of deployable component software, including descriptors for packages, interfaces, configurations, assemblies and implementations. The toplevel element is the PackageConfiguration .	Component Management Model: Repository Manager interface, which manages descriptive information about Component Software. Key operations include: <ul style="list-style-type: none"> ◆ Install Package from URL into Repository, with label ◆ Configure package, with label ◆ Retrieve package configuration info by label or top level interface UID 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, <i>possibly</i> parsing, validating, authenticating etc., and creates a default configuration for the package in the repository. Configuration tool stores settings referring to a package, <i>optionally</i> after retrieving package information for property validation. Planning tool retrieves information <i>in IDL data form</i> 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 toplevel information is the Domain .	Target Management Model: Interface to Target Manager which manages Domain information, either offline (simply parsed from private XML) or online. Needs to allow for efficient static vs. dynamic information. Key methods: <ul style="list-style-type: none"> ◆ Get base info (to allow planning tool to do preprocessing/caching of static stuff). ◆ Get current info (to plan based on dynamic 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 Target Manager would know and supply dynamic information collected from nodes. A Target Manager would initially read provided target description from XML files, and then provide the information using the data model.
Execution	Execution Data Model of decisions configuring and connecting and locating component software on a target. This is the Deployment Plan .	Target Execution Model: Interface to the Execution Manager , which represents the runtime environment for execution of component software on the target according to the plan. Key methods: <ul style="list-style-type: none"> ◆ Prepare for execution, using plan, returning “factory” reference (Application Manager) ◆ Launch based on factory, returning factory identifier and component reference. ◆ Lifecycle control. 	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.

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 **PackageConfiguration**, **Domain** and **DeploymentPlan**. The second column lists the three top-level management interfaces, **RepositoryManager**, **TargetManager** and **ExecutionManager**. Each of these six 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.

4.2 Model Diagram Conventions

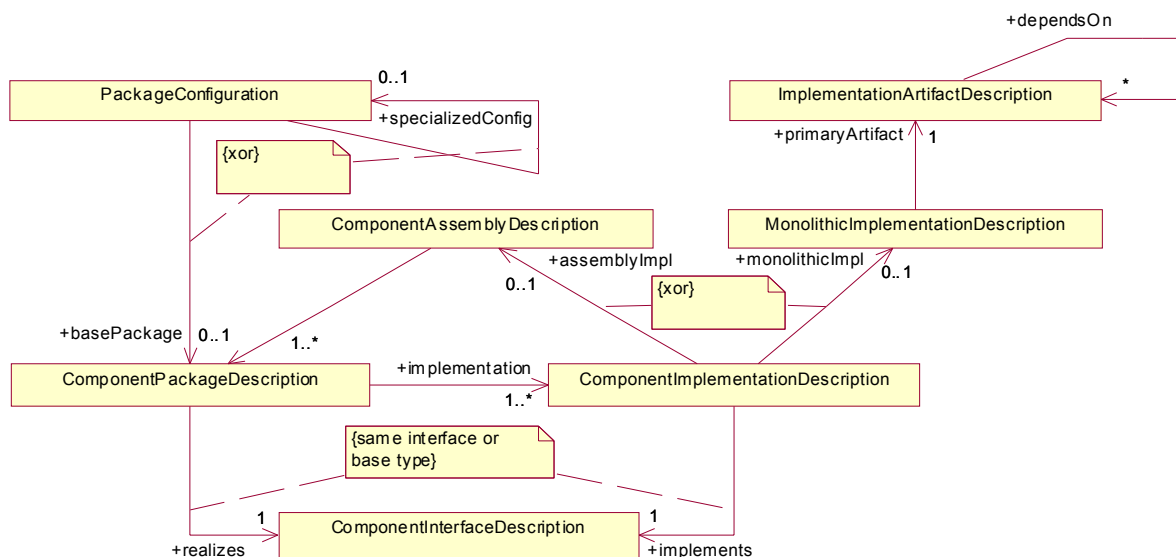
If, in a UML diagram, a class's attribute and operation compartments are suppressed, then this class is elaborated elsewhere. In this case, the diagram might also not show all of the class' associations. However, if a class is shown to have only an attribute or an operation compartment, then this signifies 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 are made explicit where ever 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.

4.3 Component Data Model

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.

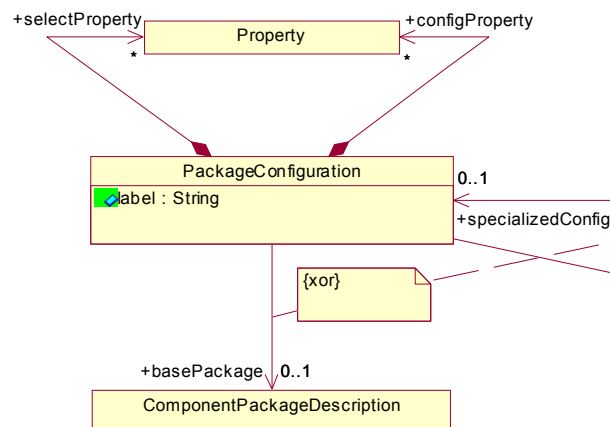


The above is an overview of the Component Data Model and represents the information about installed and configured packages provided by the **RepositoryManager**. Details about each class will be presented in the following sections:

Any	Page	24
AssemblyConnectionDescription	Page	15
ComponentAssemblyDescription	Page	13
ComponentExternalPortEndpoint	Page	16
ComponentImplementationDescription	Page	12
ComponentPackageDescription	Page	11
ComponentPackageReference	Page	15
ComponentPropertyMapping	Page	18
ComponentInterfaceDescription	Page	21
ComponentPortDescription	Page	22
ExternalReferenceEndpoint	Page	17
ImplementationArtifact	Page	21
ImplementationArtifactDescription	Page	20
MonolithicImplementationDescription	Page	19
SubcomponentInstantiationDescription	Page	14
SubcomponentPortEndpoint	Page	17
SubcomponentPropertyReference	Page	18
PackageConfiguration	Page	10
Property	Page	23
Requirement	Page	23

PackageConfiguration

Description



A **PackageConfiguration** describes one configuration of a component package. It either specializes another **PackageConfiguration** or is directly based on a **ComponentPackageDescription**. A

PackageConfiguration has 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 properties are used to influence deployment decisions by matching them against implementation-specific selection properties in the **ComponentImplementationDescription**.

Attributes

- **label: String** Unique label for this **PackageConfiguration**.

Associations

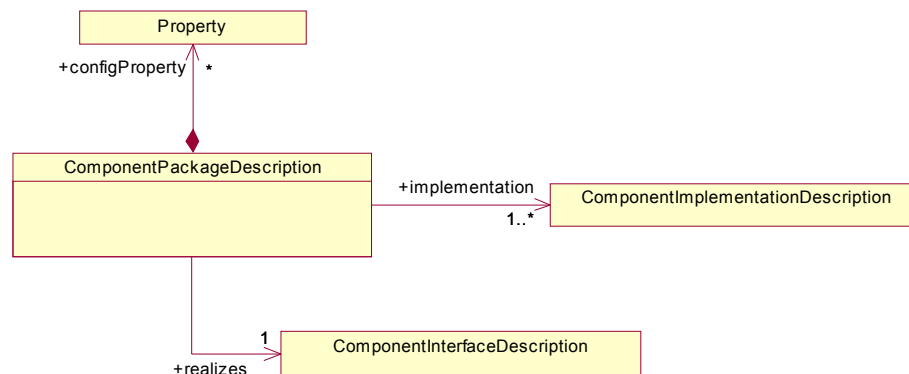
- **specializedConfig**: **PackageConfiguration** [0..1]
Links to a **PackageConfiguration** that is specialized by this **PackageConfiguration**.
- **basePackage**: **ComponentPackageDescription** [0..1]
Links to a **ComponentPackageDescription** that this **PackageConfiguration** is based on.
- **selectProperty**: **Property** [*]
During planning, selection properties in a **PackageConfiguration** are matched against selection properties in a **ComponentImplementationDescription**. This enables the **PackageConfiguration** to select among implementations.
- **configProperty**: **Property** [*]
Properties to configure the application component with. Overrides default values in the **ComponentPackageDescription**.

Constraints

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

Semantics

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

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 **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

No attributes.

Associations

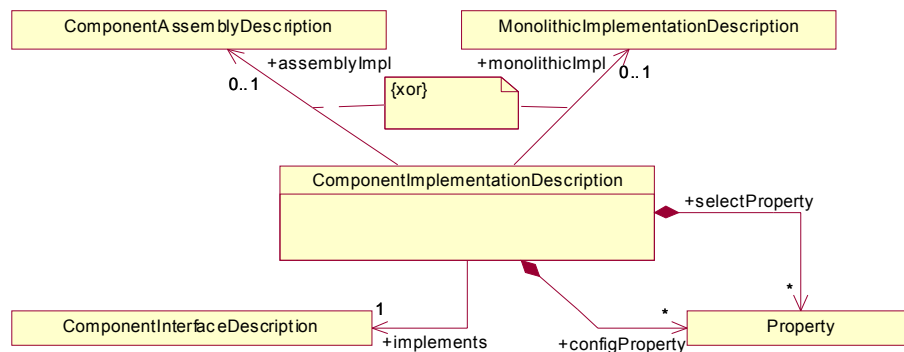
- **realizes: ComponentInterfaceDescription [1]**
A **ComponentPackageDescription** describes implementations that realize a certain component interface.
- **implementation: ComponentImplementationDescription [1..*]**
A **ComponentPackageDescription** describes multiple implementations.
- **configProperty: 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.

Semantics

Configuration properties can be overridden in a **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 **PackageConfiguration**.

ComponentImplementationDescription*Description*

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 selection properties such as “fast but coarse” or “secure”. Administrators can then select among implementations using selection properties in a **PackageConfiguration**.

Attributes

No attributes.

Associations

- **implements: 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: MonolithicImplementationDescription** [0..1]
In case of a monolithic implementation, this describes the monolithic implementation.
- **configProperty: Property** [*]
These are implementation specific configuration properties that are used to configure the component once instantiated.
- **selectProperty: Property** [*]
These are tags that a **PackageConfiguration** can match against to discriminate between implementations.

Constraints

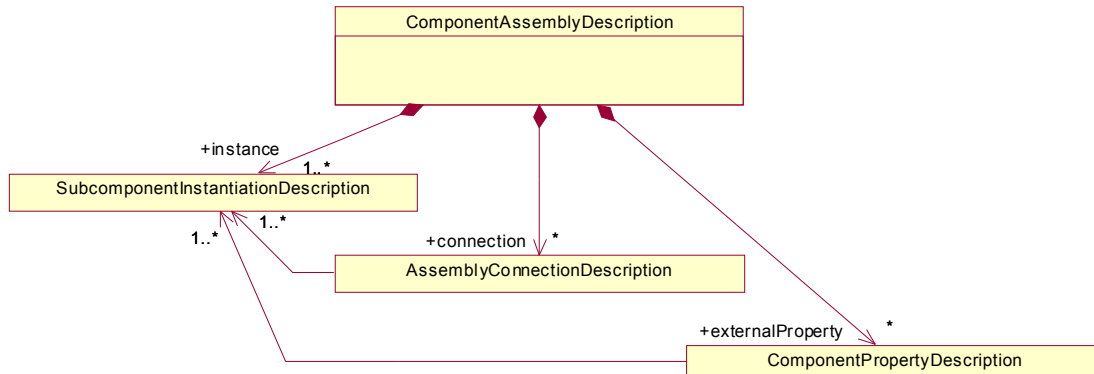
The implementation must implement an interface that is the same or derived from the interface mandated by the **ComponentPackageDescription** that this implementation is contained in. An implementation is either assembly based or monolithic, consequently there must be either a **ComponentAssemblyDescription** or a **MonolithicImplementationDescription**, but not both.

Semantics

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

ComponentAssemblyDescription

Description



In the case of an assembly based implementation, the **ComponentAssemblyDescription** contains information about sub-component instances (**SubcomponentInstantiationDescription**), connections among ports (**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: AssemblyConnectionDescription** [*]
Describes connections between ports.

- **externalProperty: ComponentPropertyMapping [*]**

Maps the external properties of the component that is implemented by the assembly to properties of subcomponent instances.

Constraints

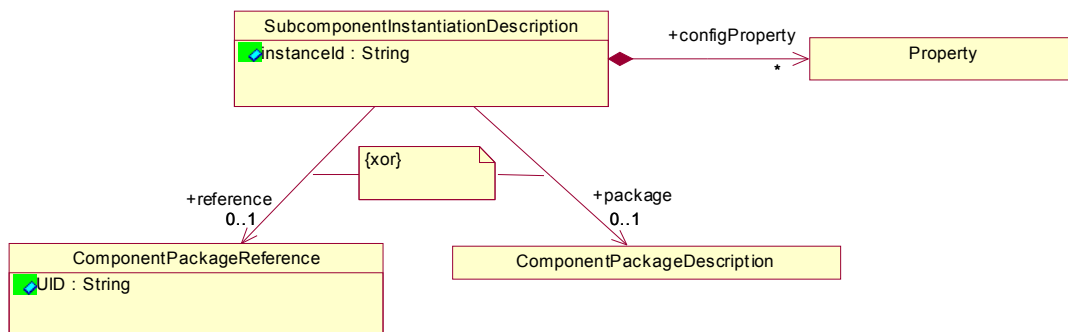
No constraints.

Semantics

An assembly is composed of components and itself implements a component, as implied by the **ComponentImplementationDescription** that this **ComponentAssemblyDescription** is contained in. 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.

SubcomponentInstantiationDescription

Description



In an assembly based implementation, the **SubcomponentInstantiationDescription** describes one instance of a sub-component and identifies it with a unique instance identifier that is used from other parts of the **ComponentAssemblyDescription** to identify this sub-component instance.

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 UID for 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

- **instancetype: String**

An identifier for this subcomponent instance that is unique within the assembly.

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: 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.

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.

ComponentPackageReference

Description

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

Attributes

- **UID: String**

Identifies a component interface by UID.

Associations

No associations.

Constraints

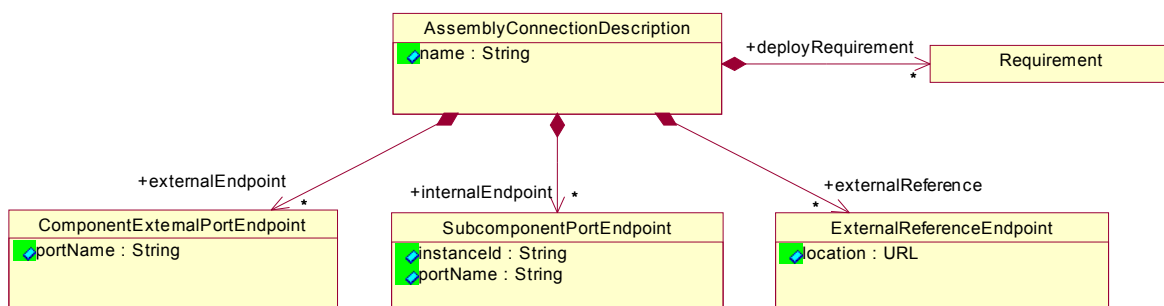
No constraints.

Semantics

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

AssemblyConnectionDescription

Description



An **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

SubcomponentPortEndpoint, which reflects a connection to the port of a subcomponent within the assembly. The **ComponentExternalPortEndpoint** reflects a connection to an external port of the component that is implemented by the assembly. The **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 name is associated with the **AssemblyConnectionDescription**. This name must be unique within each assembly. Assembly design tools might use or generate this name to visualize the connection.

Attributes

- **name: String** Identifies this connection within its assembly. May be used or generated by visual design tools.

Associations

- **deployRequirement: Requirement [*]**
These connection requirements must be satisfied by the interconnects over which the connection is routed.
- **internalEndpoint: SubcomponentPortEndpoint [*]**
Identifies a port of a component within the assembly as an endpoint of this connection.
- **externalEndpoint: ComponentExternalPortEndpoint [*]**
Identifies a port of the component that is implemented by the assembly as an endpoint of this connection.
- **externalReference: ExternalReferenceEndpoint [*]**
Identifies a location outside the assembly as an endpoint of this connection.

Constraints

No constraints.

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.

ComponentExternalPortEndpoint

Description

Identifies a port of the component that is implemented by an assembly as an endpoint of the connection described by the **AssemblyConnectionDescription** that this element is contained in.

Attributes

- **portName: String** The name of the port of the component that this assembly is implementing and that is to be an endpoint of this connection.

Associations

No associations.

Constraints

The port name must be valid for the component that this assembly is implementing.

Semantics

See above.

SubcomponentPortEndpoint

Description

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

Attributes

- **instanceId**: String References a subcomponent instance by its unique id within the assembly, from the **SubcomponentInstantiationDescription**.
- **portName**: String The name of the port of that subcomponent instance that is to be an endpoint of this connection.

Associations

No associations.

Constraints

The instance identifier must exist in the assembly, and the port name must be valid for that component.

Semantics

See above.

ExternalReferenceEndpoint

Description

Identifies a location outside the assembly as an endpoint of the connection described by an **AssemblyConnectionDescription**.

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.

Constraints

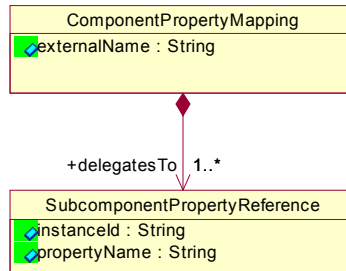
No constraints.

Semantics

See above.

ComponentPropertyMapping

Description



ComponentPropertyMapping is part of the **ComponentAssemblyDescription**. It identifies a property of the component that this assembly is implementing and the subcomponents' properties that it delegates to.

Attributes

- **externalName**: **String** The name of a property of the component that the assembly is implementing.

Associations

- **delegatesTo**: **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 component that the assembly is implementing.

Semantics

At configuration time and at runtime, if the component's property is configured, the configuration value will be delegated (propagated) to the specified subcomponent ports in the assembly.

SubcomponentPropertyReference

Description

Identifies a property of a component within the assembly that an external property of the component that the assembly implements delegates to.

Attributes

- **instanceId**: **String** References a subcomponent instance by its unique id within the assembly, from the **SubcomponentInstantiationDescription**.
- **propertyName**: **String** The name of the property of that subcomponent instance that the external property is delegated to.

Associations

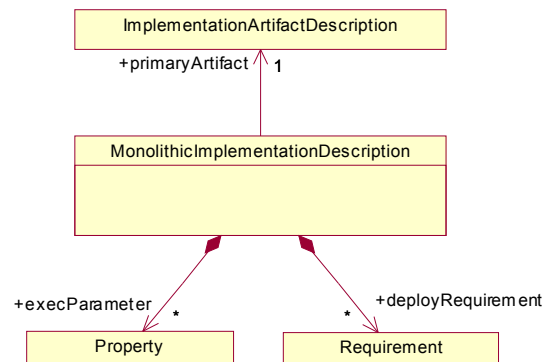
No associations.

Constraints

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

Semantics

No semantics.

MonolithicImplementationDescription*Description*

In the case of a monolithic implementation, the **MonolithicImplementationDescription** describes the artifacts that are involved in this implementation. It references a primary implementation artifact (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).

Attributes

No attributes.

Associations

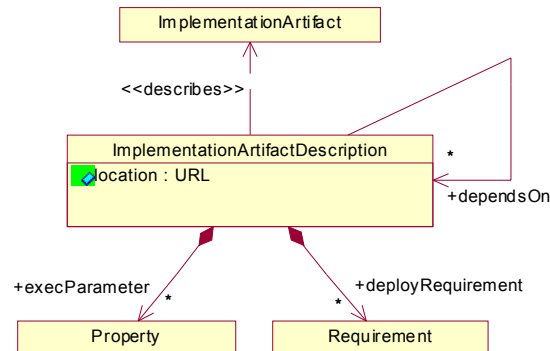
- **execParameter: Property** [*] Execution parameters that are passed to the target environment.
- **deployRequirement: Requirement** [*] Requirements that are matched against node resources during planning.
- **primaryArtifact: ImplementationArtifactDescription** [1] The primary implementation artifact.

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.

ImplementationArtifactDescription*Description*

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

Attributes

- **location: URL** The URL that can be used to access the implementation artifact.

Associations

- **dependsOn: ImplementationArtifactDescription [*]** References other **ImplementationArtifactDescription** elements for implementation artifacts that this implementation artifact depends on.
- **execParameter: Property [*]** Execution parameters with hints to the target environment about the execution of this implementation artifact.
- **deployRequirement: Requirement [*]** Requirements that are matched against node resources.

Constraints

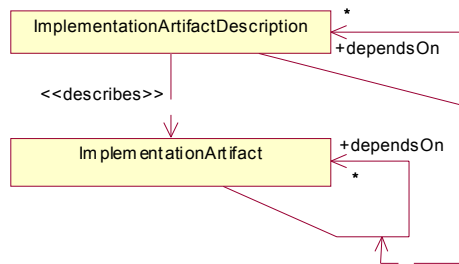
No constraints.

Semantics

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

ImplementationArtifact

Description



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

Attributes

No attributes.

Associations

No associations.

Constraints

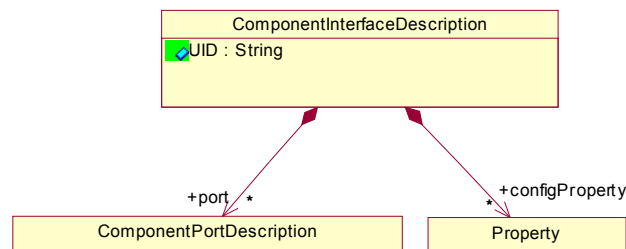
No constraints.

Semantics

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

ComponentInterfaceDescription

Description



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

- **UID: String** Uniquely identifies the component interface.

Associations

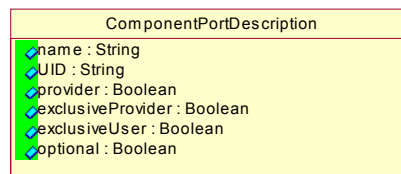
- **port: ComponentPortDescription** [*] Describes the ports of this component interface.
- **configProperty: Property** [*] Identifies the configurable properties of a component interface.

Constraints

No constraints.

Semantics

No semantics.

ComponentPortDescription*Description*

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.
- **UID: String** The type of the port. Ports that are endpoints of a connection must support the same type.
- **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

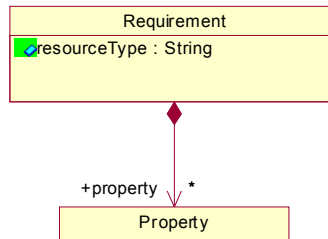
No associations.

Constraints

No constraints.

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.

Requirement*Description*

Requirement is used in the **MonolithicImplementationDescription**, **ImplementationArtifactDescription** and the **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

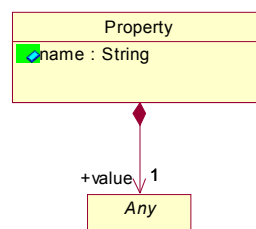
- **properties: Property [*]** Properties associated with the resource.

Constraints

No constraints.

Semantics

No semantics.

Property*Description*

A **Property** has a name and a value. It is used to carry named and typed values in various places.

Attributes

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

Associations

- **value: Any [1]** The typed value.

Constraints

No constraints.

Semantics

No additional semantics.

Any*Attributes*

No attributes.

Associations

No associations.

Constraints

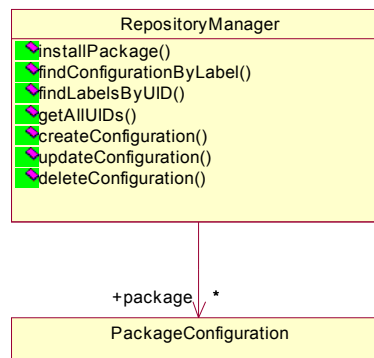
No constraints.

Semantics

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

4.4 Component Management Model

RepositoryManager

Description

A **RepositoryManager** manages component data. It maintains a collection of **PackageConfiguration** elements. Package installation results in a new **ComponentPackageDescription** represented by a **PackageConfiguration** with an empty set of properties. **PackageConfigurations** are identified by labels that are unique within the repository. The **RepositoryManager** can provide a list of all package configuration labels that support a given component interface's UID, and a list of all UIDs. **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** (label: **String**, location: **URL**): **void**

Installs a package in the repository, assigning the given label to the new **PackageConfiguration**.

- **findConfigurationByLabel** (label: **String**): **PackageConfiguration**
Locates a **PackageConfiguration** by label.
- **findLabelsByUID** (UID: **String**): **Sequence(String)**
Finds all labels for configurations for packages that support the given interface identifier. Returns a sequence of labels.
- **getAllUIDs** (): **Sequence(String)**
Returns a sequence of all UIDs for which packages are available.
- **createConfiguration** (nlabel: **String**, blabel: **String**, sp: **Sequence(Property)**, cp: **Sequence(Property)**): **void**
Creates a new **PackageConfiguration** based on an existing configuration, extending and overriding the base's selection and configuration properties.
- **updateConfiguration** (label: **String**, sp: **Sequence(Property)**, cp: **Sequence(Property)**): **void**
Updates an existing **PackageConfiguration**, extending and replacing its selection and configuration properties.
- **deleteConfiguration** (label: **String**, deletePackage: **Boolean**): **void**
Deletes the **PackageConfiguration** that is referenced by label and all other **PackageConfiguration** elements that are based on it. If this is the last **PackageConfiguration** for a component package, and if the **deletePackage** parameter is set to true, then the package is also removed from the repository. If this is the last **PackageConfiguration** for a component package, and if the **deletePackage** parameter is false, then deletion of the **PackageConfiguration** fails.

Associations

- **package**: **PackageConfiguration** [*]

A **RepositoryManager** manages a number of package configurations.

Constraints

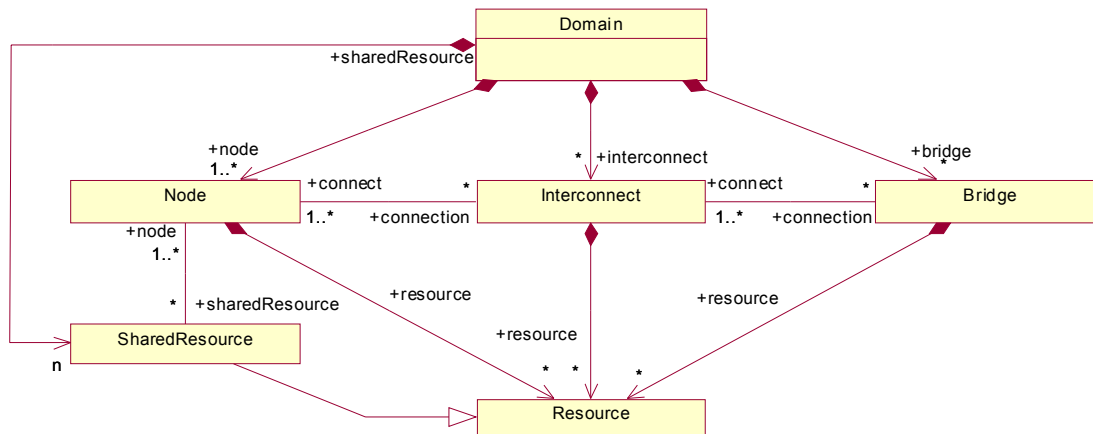
No constraints.

Semantics

No additional semantics.

4.5 Target Data Model

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 **Domain**. A **Domain** is composed of **Node**, **Interconnect**, **Bridge** and **SharedResources** 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:

Any	Page	24
Bridge	Page	29
Domain	Page	27
Interconnect	Page	28
Node	Page	27
Resource	Page	30
ResourceProperty	Page	31
ResourcePropertyKind	Page	31
SharedResource	Page	30

Domain

Description

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

Attributes

No attributes.

Associations

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

Constraints

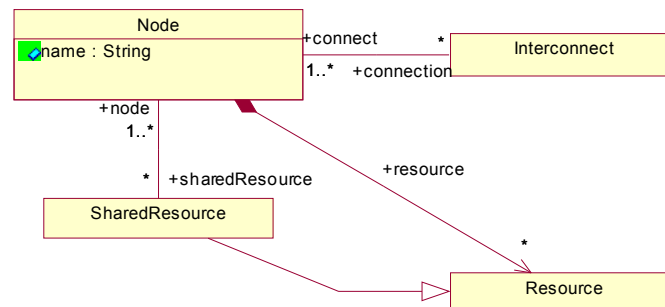
No constraints.

Semantics

No additional semantics.

Node

Description



A **Node** has a name that must be unique within the **Domain**. 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** A node has a name.

Associations

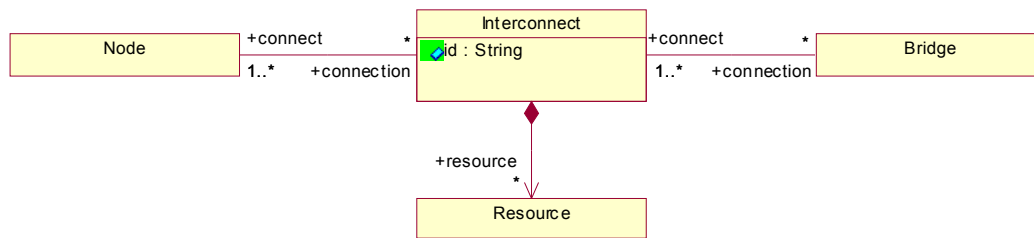
- **connection**: **Interconnect** [*] A node may be connected to interconnects.
- **resource**: **Resource** [*] A node may have resources.
- **sharedResource**: **SharedResource** [*] A node may have access to shared resources.

Constraints

The name of the **Node** must be unique within the **Domain**.

Semantics

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

Interconnect*Description*

An **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 **SubcomponentConnectionInformation**) 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

- **id: String** An identifier that is unique within the domain.

Associations

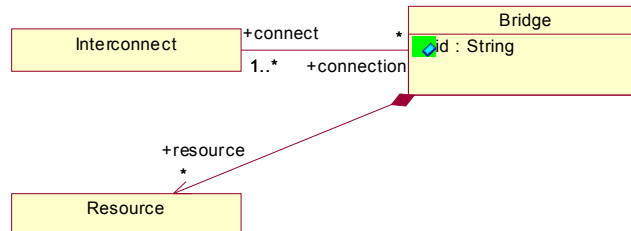
- **connect: Node [1..*]** The nodes that this interconnect provides a connection in between.
- **connection: Bridge [*]** The bridges that provide connectivity to other interconnects.
- **resource: Resource [*]** Interconnects have resources.

Constraints

The identifier must be unique within the domain.

Semantics

An interconnect's resources are matched against connection requirements.

Bridge*Description*

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

- **id: String** An identifier that is unique within the domain.

Associations

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

Constraints

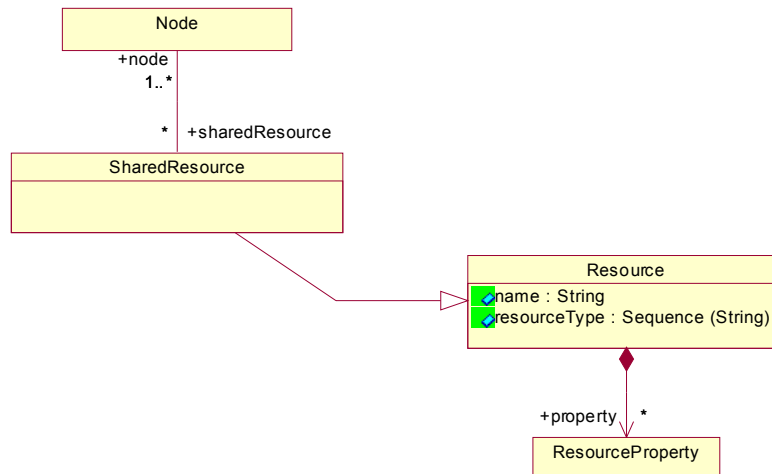
The identifier must be unique within the domain.

Semantics

A bridge's resources are matched against connection requirements.

Resource

Description



Resource elements express **Node**, **Interconnect** and **Bridge** features within the target environment. They are matched against implementation requirements at planning time.

Attributes

- **name**: **String** Uniquely identifies the resource.
- **resourceType**: **Sequence(String)** The resource types supported by this resource.

Associations

- **property**: **ResourceProperty** [*] Properties associated with this resource.

Constraints

There must be at least one element in the **resourceType** sequence attribute. The name must be unique among **Resource** elements in the same container (**Node**, **Interconnect**, **Bridge** or **Domain**).

Semantics

Several properties are associated with a resource. The type of the resource is matched against the type of a **Requirement**. For matching types, the requirement's properties will then be matched against the resource's properties.

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 attributes.

Associations

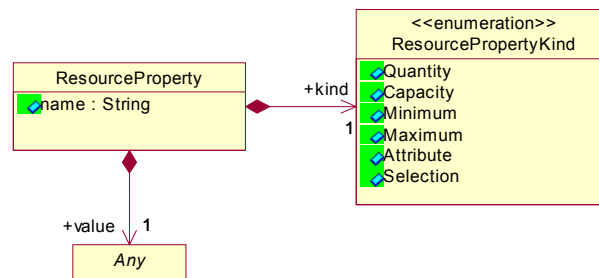
- **nodes**: **Node** [1..*] The nodes that have access to this **SharedResource**.

Constraints

The name of the **SharedResource** must be unique among all **SharedResource** elements within the **Domain**.

Semantics

Same as for **Resource**.

ResourceProperty*Description*

Describes a specific property of a **Resource** or **SharedResource**. It contains a **ResourcePropertyKind** that classifies the **ResourceProperty** and has implications on the type of the value and the comparison between the **ResourceProperty** and a required **Property**.

Attributes

- **name**: **String** The name of the property.
- **value**: **Any** The typed value.

Associations

- **kind**: **ResourcePropertyKind** [*] The class of the property.

Semantics

ResourceProperty elements are matched against the **Property** elements within a **Requirement** at planning time. They describe attributes and capacities of some element in the target environment. The **name** attribute of the **ResourceProperty** must match the **name** attribute of the **Property** it is compared against. Matching the values will be discussed as part of the **ResourcePropertyKind** semantics.

ResourcePropertyKind*Description*

Classifies a **ResourceProperty**. Each **ResourcePropertyKind** identifies a specific way to match requirements against resources. The kind of **ResourcePropertyKind** implies the types of the values contained in **ResourceProperty** and **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 **ResourceProperty**, and "requirement" refers to the property element of the **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 requirement does not have an associated value (type "void"), 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.

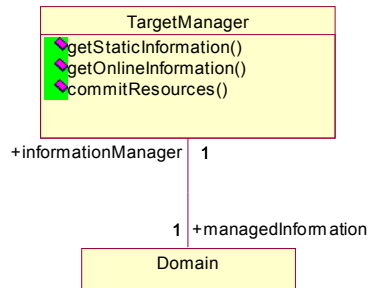
The Quantity and Attribute kinds are redundant, but included here to account for these common use cases. (Quantity is equivalent to a Capacity that is required in amounts of one, and Attribute is a subset of Selection.)

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.

4.6 Target Management Model

TargetManager

Description



The **TargetManager** provides information about the **Domain** using the Target Data Model. It can supply both static information (with no resource tracking) and online information (tracking resource usage). The **TargetManager** also provides an operation to commit resources. Note that this specification limits the features of the **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

- **getStaticInformation ()**: **Domain** Returns static information about the domain, with resources at their full capacity.
- **getOnlineInformation ()**: **Domain** Returns online information about the domain; resources will reflect their remaining capacity.
- **commitResources (plan: DeploymentPlan)**: **void** Commits resources that are used by the instantiation of an application from a deployment plan.

Associations

- **managedInformation**: **Domain** [1] A **TargetManager** manages information about a single **Domain**.

Constraints

No constraints

Semantics

A **TargetManager** can be online or offline. An online **TargetManager** monitors a concrete target environment and is able to supply up-to-date resource information. An offline **TargetManager** only tracks resources as committed via **commitResources**; this is useful for planning in advance.

The acquisition of resource information by an online **TargetManager** is a quality of implementation issue.

Transaction support may be necessary to avoid race conditions in online planning – otherwise resources might be committed elsewhere while planning.

4.7 Execution Data Model

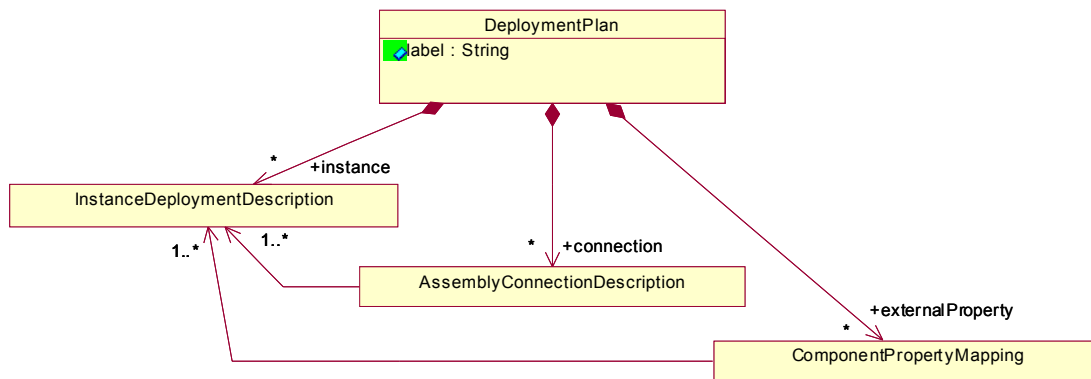
Before deployment can occur, decisions must be made about the implementations to select (if multiple implementations exist in a package) and where to deploy each monolithic component implementation. All information about an application's deployment is collected in a **DeploymentPlan**. This plan can be used transiently (i.e. executed right away), or it may be stored to avoid the overhead of planning in the future. The **DeploymentPlan** can be used by an **ExecutionManager** to create a specific factory object for the application. A **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:

ArtifactDeploymentDescription	Page	36
AssemblyConnectionDescription	Page	15
ComponentPropertyMapping	Page	18
DeploymentPlan	Page	34
InstanceDeploymentDescription	Page	35
Property	Page	23
Requirement	Page	23

DeploymentPlan

Description



The **DeploymentPlan** contains information about which components to deploy where (**InstanceDeploymentDescription**), about interconnections between them (**AssemblyConnectionDescription**), and about the mapping of external properties. The **DeploymentPlan** is analogous to the **ComponentAssemblyDescription** in the Component Data Model and reuses the **AssemblyConnectionDescription** and **ComponentPropertyMapping** elements. In fact, the **DeploymentPlan** can be seen as a flattened assembly (without recursion) that includes information about where and how to instantiate each component. In the flattened 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 components that have a monolithic implementation. **InstanceDeploymentDescription** describes how and where to instantiate a component from its monolithic implementation.

Attributes

- **label: String**

Users may optionally assign a label to a **DeploymentPlan**.

Associations

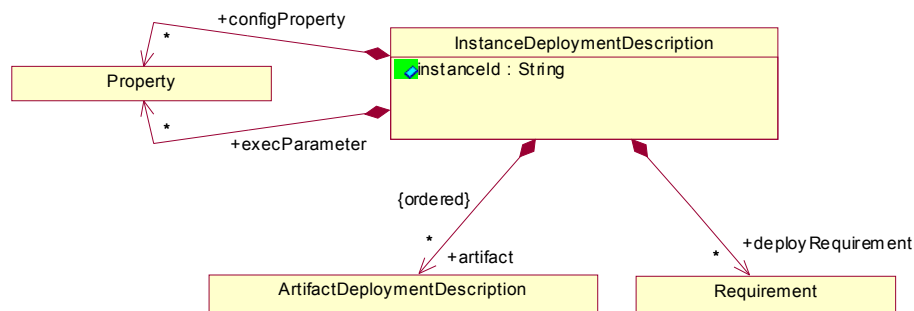
- **instance**: **InstanceDeploymentDescription** [*]
Component instances that are to be created.
- **connection**: **AssemblyConnectionDescription** [*]
Connections that are to be made between the component instances, the application's external ports, or external locations.
- **externalProperty**: **ComponentPropertyMapping** [*]
Maps the application's external properties to properties of component instances.

Constraints

No constraints.

Semantics

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

InstanceDeploymentDescription*Description*

InstanceDeploymentDescription contains the information that is necessary in order to deploy a single component instance from a monolithic implementation. The resulting component instance is assigned a unique instance identifier that will be used to reference this instance within the **DeploymentPlan**. The **InstanceDeploymentDescription** contains a number of **ArtifactDeploymentDescription** elements for all artifacts that have to be installed for the deployment to work. This reflects the top level implementation artifact from a **MonolithicImplementationDescription** and a depth-first traversal of its dependencies. Note that a deployment planning tool would possibly generate the **instanceId** string from a concatenation of the **instanceId** elements in the hierarchical assemblies. This would provide traceability of the flattening similar to the techniques used in hardware design tools.

Attributes

- **instanceId**: **String**
A unique identifier for this component instance, used by the **AssemblyConnectionDescription** and the **ComponentPropertyMapping**.

Associations

- **artifact**: **ArtifactDeploymentDescription** [*]
The artifacts that need to be installed on a node in order to instantiate this component.

- **deployRequirement**: **Requirement** [*]
Requirements copied from the **MonolithicImplementationDescription**.
- **execParameter**: **Property** [*]
Execution parameters copied from the **MonolithicImplementationDescription**.
- **configProperty**: **Property** [*]
Properties used to configure the component instance with after instantiation.

Constraints

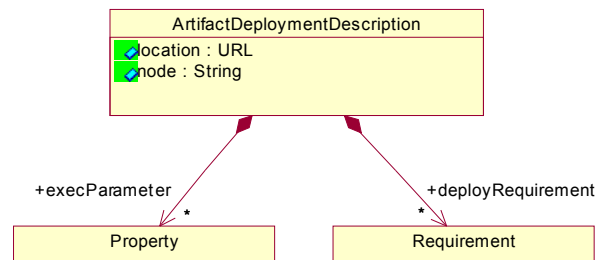
The elements in the **artifact** association are ordered so that a depended-on artifact is installed before the implementation artifacts that depend on it.

Semantics

The deployment requirements carry information about the resources used by this implementation, so that they can be committed by the **TargetManager**. Configuration properties reflect the final configuration, respecting overloading between **SubcomponentInstantiationDescription**, **ComponentImplementationDescription**, **ComponentPackageDescription** and **PackageConfiguration**.

ArtifactDeploymentDescription

Description



ArtifactDeploymentDescription describes the installation of a single implementation artifact on a node as part of component instantiation. It contains an URL pointing to the **ImplementationArtifact** and the name of the node where the implementation artifact is to be installed. Properties and requirements are copies of the properties and requirements from the **ImplementationArtifactDescription**.

Attributes

- **location**: **URL**
The URL where the artifact can be loaded from. Copied from the **ImplementationArtifactDescription**.
- **node**: **String**
The name of the node where the artifact is to be installed.

Associations

- **execParameter**: **Property** [*]
Execution parameters, copied from the **ImplementationArtifactDescription**.
- **deployRequirement**: **Property** [*]
Deployment requirements, copied from the **ImplementationArtifactDescription**.

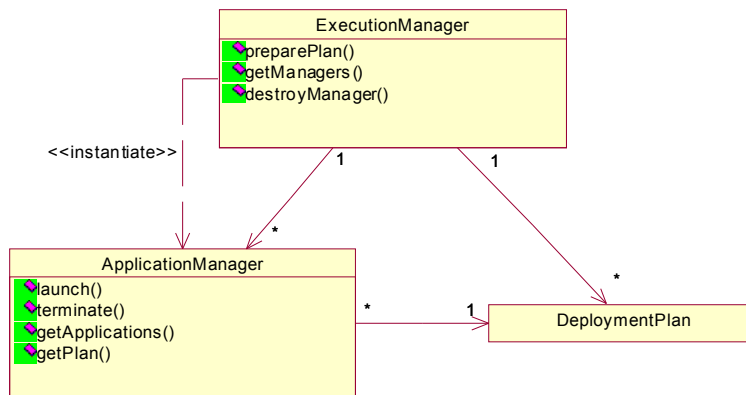
Constraints

No constraints.

Semantics

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

4.8 Execution Management Model

**ExecutionManager***Description*

The **ExecutionManager** manages the execution of applications.

Operations

- **preparePlan** (plan: **DeploymentPlan**): **ApplicationManager**
Create an application manager (factory) from a deployment plan.
- **destroyManager** (manager: **ApplicationManager**): **void**
Terminate an application manager and free all associated resources.
All running applications are terminated as well.
- **getManagers** (): **Sequence(ApplicationManager)**
Return a list of all active application managers.

Associations

- **theApplicationManager**: **ApplicationManager** [*]
An **ExecutionManager** instantiates **ApplicationManagers** instances.
- **theDeploymentPlan**: **DeploymentPlan** [*]
The deployment plans associated with the existing active **ApplicationManager** instances.

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.

ApplicationManager

Description

The **ApplicationManager** can then be used to first launch and later to terminate the application. To support introspection, it also includes an operation to retrieve the plan that was used to create the factory and the list of currently running applications.

Operations

- **launch ()**: **Application** Executes the application from the plan and returns a handle to the application component.
- **terminate (app: Application)**: **void** Terminates a running application.
- **getApplications ()**: **Sequence (Application)** Returns a list of all applications that have been launched from this **ApplicationManager** and that are still executing.
- **getPlan ()**: **DeploymentPlan** Returns the **DeploymentPlan** associated with this **ApplicationManager**.

Associations

- **theDeploymentPlan**: **DeploymentPlan** [1] The **DeploymentPlan** that this **ApplicationManager** is based on.

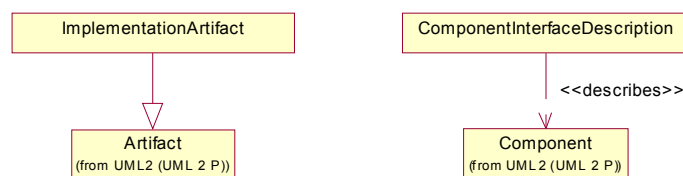
Constraints

Depending on the plan and whether it was based on static or online resource data, launching multiple applications from the same **ApplicationManager** in parallel might fail.

Semantics

The semantics of launching depend on what was done during preparation.

4.9 Relations to Other Standards



The **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 **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. **Artifact** and **Component** will 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.

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. The only requirement imposed by this specification is that all actors exist, and that their common behavior adheres to the rules described in this chapter.

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.

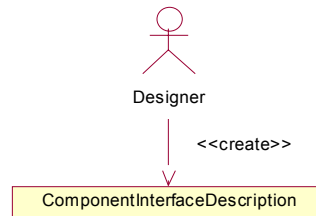
5.1 Development Actors Overview

The development of a component implementation involves the roles of Designer, Implementor, Assembler and Packager. The Designer creates an interface specification. Implementors 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.

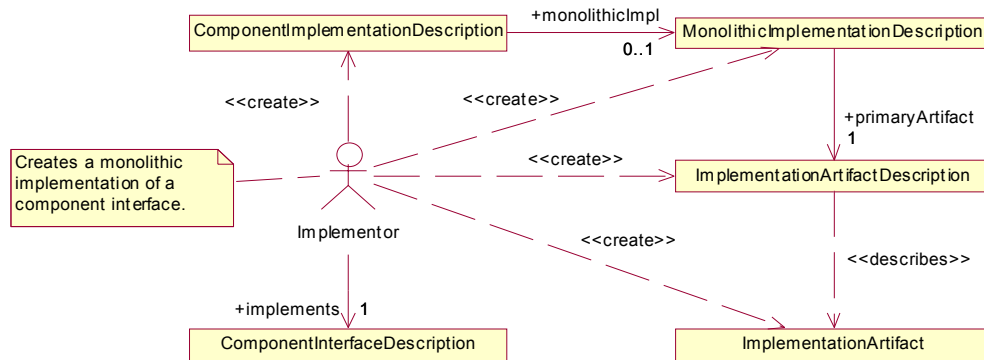
The above paragraph implies a bottom-up approach to component development, but that is not necessarily true, the flow of information can be reversed. An Implementor or Assembler can also work “downwards” from an existing component package in order to add new implementations to the package. An Assembler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

5.2 Designer



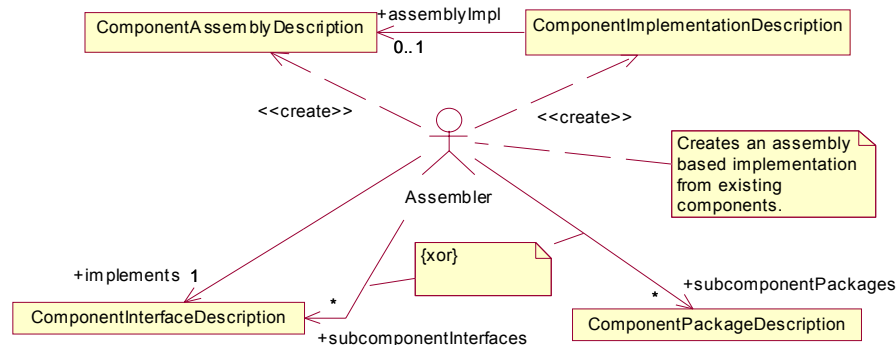
The Designer creates an interface specification and generates a **ComponentInterfaceDescription** to describe the component interface, including its ports. Designers usually create other documents as well, such as PSM-specific interface descriptions (e.g. IDL files), behavioral models and system specifications,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

5.3 Implementor



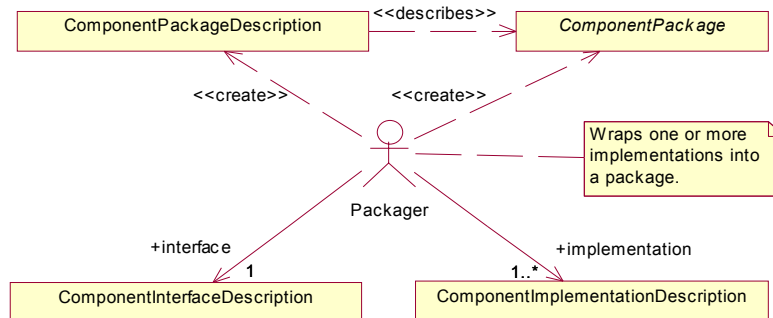
The Implementor creates a monolithic implementation that satisfies a specific component interface. The Implementor reads the Designer’s **ComponentInterfaceDescription** and creates an implementation contained in one or more implementation artifacts. For each **ImplementationArtifact**, the Implementor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

5.4 Assembler



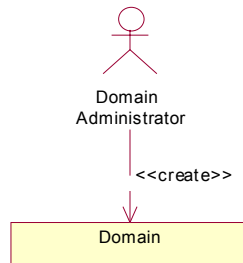
The Assembler creates an assembly based implementation of a specific component interface, using existing components as building blocks. The Assembler uses either interface descriptions for subcomponents from **ComponentInterfaceDescription** elements (expecting implementations for such interfaces to 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.

5.5 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.

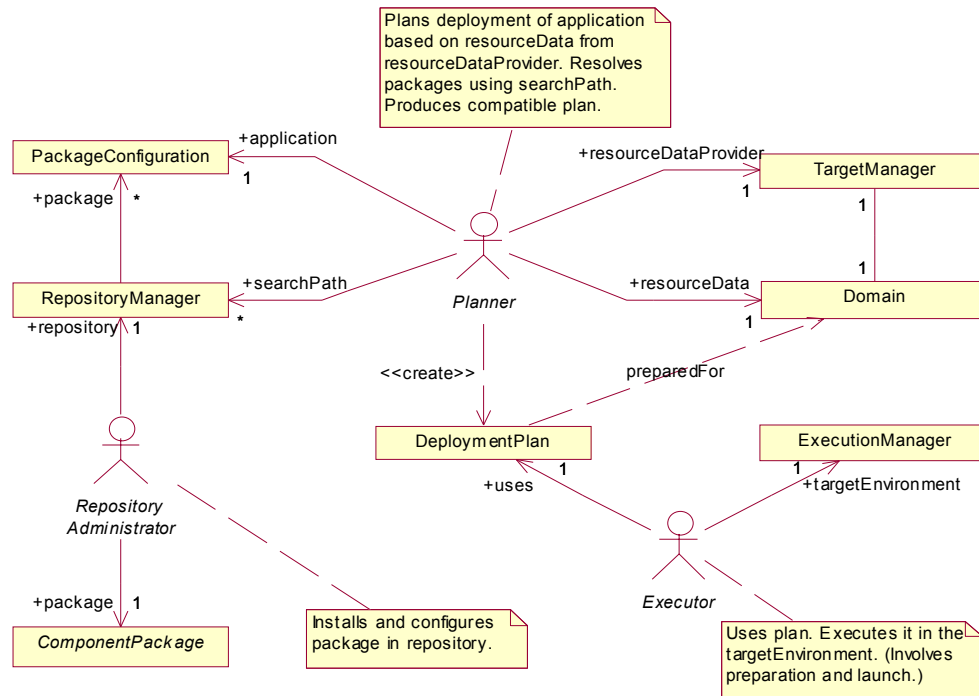
5.6 Domain Administrator



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.

5.7 Deployment Actors Overview



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.

5.8 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**.

5.9 Planner

The Planner supports planning the deployment of an application.

The Planner has access to a specific **PackageConfiguration** via a repository reference and a label: the Planner uses the **findConfigurationByLabel** 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 interface UID 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 static or online resource data from a **TargetManager** using either the **getStaticInformation** or **getOnlineInformation** 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.

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 selection properties from the **PackageConfiguration** against the selection properties in this **ComponentImplementationDescription** (see below). If they are not compatible, return to step 3 and continue iterating over implementations for 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.

Matching Selection Properties

ToDo — Explain how to match selection properties. Try to figure that out for ourselves first.

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 artifact 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.

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.

Matching a Resource against a Requirement

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

Each **Property** is then matched against the **ResourceProperty** according to the rules set forth for the kind of **ResourceProperty**, as described in the documentation for **ResourcePropertyKind**, 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**.

5.10 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 **ApplicationManager** reference that is returned.

To launch an application, the Executor remembers the **ApplicationManager** reference that was the result of preparation, and calls the **launch** operation.

6.1 About the SCA

The Software Communications Architecture (SCA) specification is published by the Joint Tactical Radio Systems (JTRS) Joint Program Office (JPO). The SCA establishes an implementation-independent framework with baseline requirements for the development of JTRS software configurable radios.

The SCA is a CORBA-based component architecture for the embedded systems domain that bears a resemblance to the CORBA Component Model. In the SCA, components (of type **Resource**) are interconnected via ports. The target environment is defined by nodes with computational abilities (of type **ExecutableDevice**) and generic hardware (of type **Device**). The target environment is controlled by the **DomainManager**, **DeviceManager** and the **FileManager**.

6.2 Introduction

This section describes the mapping of the platform-independent model for Deployment and Configuration to the Software Communications Architecture.

Several management concepts that appear in the platform-independent model already exist in the SCA. Therefore, some of the management interfaces and their operations are mapped to existing SCA interface definitions.

Because of time constraints, this section is not complete yet. For the time being, this section identifies ideas and concepts that will guide the mapping. These concepts will be refined, and specific rules will be supplied with later revisions of this document.

6.3 Meta-Concepts

The meta-concept of a component in the PSM is mapped to **Resource** in the SCA. A **Resource** in the SCA has properties and ports.

The meta-concept of **ImplementationArtifact** is mapped to a file. This PSM still treats files as opaque. Agreement between the author of an implementation and the **ExecutionManager** over the contents of an implementation artifact is assumed. This agreement, or “contract,” is expressed in terms of artifact requirements, node resources and execution parameters.

The meta-concept of a package is mapped to a ZIP file that includes implementation artifacts and descriptors. XML Descriptor files are placed in the top level “meta-inf” directory in the ZIP file.

6.4 Use Case Mapping

The **DomainManager** in SCA encompasses all management interfaces from the platform independent model: it acts as a **RepositoryManager** that applications can be installed in; it acts as a **TargetManager** by keeping track of available **Devices**, and it acts as an **ExecutionManager** by enabling the execution of applications.

Installation of a component package into a repository maps to the **installApplication** method of the **DomainManager**. The result of installation is an **ApplicationFactory**.

Configuration of a component package cannot be done within the repository. However, an initial configuration can be passed along to the create operation of the Application Factory when instantiating the application.

Deployment planning is done by the **ApplicationFactory** as part of application creation. The **ApplicationFactory** retrieves target information from **Device** instances via their **DeviceManager** and the **DomainManager**. The planning process can be influenced by passing device assignments to the create operation of the ApplicationFactory.

Preparation of a planned deployment is also part of what happens in the create operation of the **ApplicationFactory** interface. The result of preparation is an **Application**.

Launch of a prepared deployment is again part of the create operation of the **ApplicationFactory** interface.

An application is destroyed when the Application’s **releaseObject** operation is called.

6.5 Mapping Ideas

The SCA can use almost the same mappings to XML schemas and IDL as the platform specific model for CCM. Differences exist in the Component Interface Description, as the description for an SCA Resource is more limited than for a CORBA component.

The current SCA’s facilities that relate to deployment can be implemented on top of the management interfaces easily.

The **DomainManager** implements the **installApplication** operation by invoking the **install** operation on the **RepositoryManager**, and then use the new configuration’s label to instantiate the **ApplicationFactory**; the label becomes the **softwareProfile** attribute.

The **ApplicationFactory** includes a planner. When the create operation is called, the **ApplicationFactory** retrieves the **PackageConfiguration** from the repository and online target information from the **TargetManager**. The initial configuration that is an input parameter to the **create** operation is applied to the **PackageConfiguration**. The **ApplicationFactory** then matches requirements against resources as usual, re-

specting the device assignment hints that are an input parameter to the create operation. After planning is complete, the **ApplicationFactory** immediately contacts the **ExecutionManager** to prepare the plan, and it then contacts the **ApplicationManager** to launch the application.

The mapping to the CCM platform introduced an application proxy that implemented the component interface and that then delegated to the appropriate subcomponent. In the SCA mapping, this proxy always exists, implementing the **Application** interface. This proxy is responsible for registering itself with the **DomainManager**.

Note that the **preparePlan** operation of the **ExecutionManager** can be called not only from an **ApplicationFactory**, but also directly, e.g. in the case of offline planning, when the plan was created based on static resource data.

Because of the tight ties between the existing **DomainManager** and the new **TargetManager** and **ExecutionManager** interfaces, a Core Framework implementation might decide to implement all three interfaces in a single implementation.

In the Target Data Model, an **ExecutableDevice** instance appears as **Node**, and a **Device** that is not an **ExecutableDevice** appears as a **Resource** or **SharedResource**. In the SCA, the Domain is not static but mutable. Adding or removing devices via the **DomainManager** or any **DeviceManager** must be reflected by the **TargetManager**.

7.1 Introduction

This section describes the mapping of the platform-independent model for Deployment and Configuration to the CORBA Component Model platform. It is supposed to be a replacement for the Packaging and Deployment chapter of the CCM specification in CORBA 3.0.

Data models are used in two different ways, first for persistent storage of information, and second for representing the data at runtime. For persistent storage, 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 *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 models are mapped to IDL data structures.

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

Because of time constraints, this section is not complete yet, lacking XML schema and IDL files. Instead, it explains the ideas and rules that will be used to auto-generate these files from the platform independent model by the way of stereotyping classes appropriately and then using rules set forth in the UML profile for CORBA. We are evaluating what other adjustments to the model are necessary, if any, so that code generation can happen automatically.

7.2 Meta-Concepts

The meta-concept of **Component** in the PIM is mapped to both components and homes for the CCM platform. Components in CCM have an interface, properties (attributes) and ports.

Viewing homes as a kind of component allows this model to deploy homes (by themselves or as part of an assembly). Applications can then use the home to create component instances at runtime. This supports the full feature set of CCM. Homes have an interface and properties (attributes), but no ports, so in their interface description that is part of the component data model, the set of ports will be empty.

Components and homes have an interface of their own and may support additional interfaces. If a connection is to be made to the component's or home's interface (that is seen as inheriting all supported interfaces), then the port name of the **ComponentExternalPortEndpoint** or **SubcomponentPortEndpoint** class is set to the magic word "main".

Repository Ids will be used as UUIDs to identify component and port interfaces.

The meta-concept of **ImplementationArtifact** is mapped to a file. This PSM still treats files as opaque. Agreement between the author of an implementation and the **ExecutionManager** over the contents of an implementation artifact is assumed. This agreement, or "contract," is expressed in terms of artifact requirements, node resources and execution parameters.

The meta-concept of a package is mapped to a ZIP file that includes implementation artifacts and descriptors. XML Descriptor files are placed in the top level "meta-inf" directory in the ZIP file.

7.3 Component Data Model

Mapping to XML

The Component Data Model will be mapped to several XML schemas, so that instances of the model (XML files) will describe pieces of an implementation relating to the different roles of the humans or tools that handle the data.

One schema will describe the Component Interface Descriptor containing data from the **ComponentInterfaceDescription**. The interface designer will create this file along with the IDL for the component interface. A Component Interface Descriptor can generally be auto-generated from the component's or home's IDL.

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 Component Package Descriptor contains data from the **ComponentPackageDescription**. It is created by the person that packages up one or more implementations into a package. The Component Package Descriptor has the same base file name as the package that it is contained in.

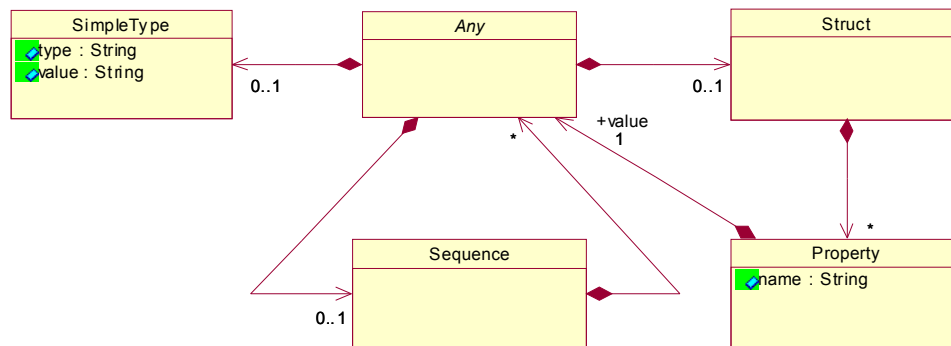
The **PackageConfiguration** class is not mapped into any schema; it is used at runtime only.

By splitting data across files, some associations in the model cross file boundaries: from **ComponentPackageDescription** to **ComponentImplementationDescription** and **ComponentInterfaceDescription**, from **ComponentImplementationDescription** to **ComponentImplementationDescription**, from **MonolithicImplementationDescription** and **ImplementationArtifactDescription** to **ImplementationArtifactDescription**, and from **SubcomponentInstantiationDescription** to **ComponentPackageDescription**.

For the XML schema, these associations will be replaced by two optional attributes “**fileinarchive**” and “**link**” with an exclusive-or relationship between them. The **fileinarchive** attribute is a string pointing to another file in the same package (ZIP file). The **link** attribute is a URL that points to the location of another file. The referenced file is an XML descriptor file of appropriate type, with the exception of the association between **SubcomponentInstantiationDescription** and **ComponentPackageDescription**. In that case, the other file can be either a Component Package Descriptor (i.e. an XML file) or a package (i.e. a ZIP file).

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

In the Implementation Artifact Descriptor, the URL attribute is also replaced by **fileinarchive** and **link** attributes. If a **fileinarchive** attribute is used, the **RepositoryManager** will substitute an URL pointing to the repository.



The abstract **Any** type is mapped to XML as above. It is used to carry arbitrary property values and must therefore be able to represent values for all IDL data types. In this mapping, an **Any** value can be either a sequence, structure or a simple type. The sequence element can also be used for arrays, the struct element can also be used for valuetypes. For simple types, the type and value are encoded as strings. Note that the type of the **Any** is implied by its context.

Filename extensions for the various descriptors are TBD.

Mapping to IDL

All classes in the Component Data Model are mapped to CORBA structures. This involves stereotyping all classes as «CORBAstruct» and using the rules set forth in the UML Profile for CORBA.

The models’s abstract **Any** type is mapped to the IDL any type.

All generated IDL is placed in the Deployment module.

7.4 Component Management Model

The **RepositoryManager** is a runtime entity, it needs to be mapped to IDL only. This is achieved by stereotyping the **RepositoryManager** class as «CORBAInterface» and by using the rules set forth in the UML Profile for CORBA.

The generated **RepositoryManager** interface is placed in the Deployment module.

7.5 Target Data Model

A **Domain** is defined by a set of three graphs. The first graph has nodes as vertices and interconnects as edges. The second graph has interconnects as vertices and bridges as edges. The third graph describes the association between nodes and shared resources. To support efficient representation of these graphs in XML and IDL, the associations between **SharedResource**, **Node**, **Interconnect** and **Bridge** are mapped to attributes that represent the associations using identifiers.

For **Node**, the **connection** association is mapped to a **connection** attribute that holds a sequence of strings; each element identifies an **Interconnect** by its **id** attribute. The **sharedResources** association is mapped to a **sharedResources** attribute that holds a sequence of strings; each element identifies a **SharedResource** by its **name** attribute.

For **Interconnect**, the **connect** association is mapped to a **connect** attribute that holds a sequence of strings; each element identifies a **Node** by its **name** attribute. The **connection** association is mapped to a **connection** attribute that holds a sequence of strings; each element identifies a **Bridge** by its **id** attribute.

For **Bridge**, the **connect** association is mapped to a **connect** attribute that holds a sequence of strings; each element identifies an **Interconnect** by its **id** attribute.

For **SharedResource**, the **nodes** association is mapped to a **nodes** attribute that holds a sequence of strings; each element identifies a **Node** by its **name** attribute.

Mapping to XML

The platform independent Target Data Model, with the above changes, is mapped to a single XML schema. Consequently, domain information will be contained in a single XML file. The filename extension of this file is TBD.

Note – 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 fine. However, the XML mapping comes at no price, so it is included to discourage incompatible extensions to the target data model.

Mapping to IDL

All classes in the Target Data Model are mapped to CORBA structures. This involves stereotyping all classes as «CORBAStruct» and using the rules set forth in the UML Profile for CORBA.

For the mapping of **SharedResource** to a CORBA structure, the inheritance from **Resource** is removed; in its place, all attributes of the **Resource** class (**name**, **resourceType**) and its associations (**properties**) are added to **SharedResource** itself.

All generated IDL is placed in the Deployment module.

7.6 Target Management Model

The **TargetManager** is a runtime entity, it needs to be mapped to IDL only. This is achieved by stereotyping the **TargetManager** class as «CORBAInterface» and by using the rules set forth in the UML Profile for CORBA specification.

The generated **TargetManager** interface is placed in the Deployment module.

7.7 Execution Data Model

Note that the associations from **AssemblyConnectionDescription** and **ComponentPropertyMapping** to **InstanceDeploymentDescription** are derived, they are realized by **instanceId** attributes in the **SubcomponentPortEndpoint** and **SubcomponentPropertyReference**, respectively; they identify a specific **InstanceDeploymentDescription** in the **DeploymentPlan**.

Mapping to XML

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

Mapping to IDL

The **DeploymentPlan** is mapped to a CORBA structure. This involves stereotyping all classes as «CORBAStruct» and using the rules set forth in the UML Profile for CORBA specification.

All generated structures are placed in the Deployment module.

7.8 Execution Management Model

ExecutionManager and **ApplicationManager** are runtime entities, they need to be mapped to IDL only. This is achieved by stereotyping both classes as «CORBAInterface» and by using the rules set forth in the UML Profile for CORBA specification.

The generated **ExecutionManager** and **ApplicationManager** interfaces are placed in the Deployment module.

Applications implement a specific component interface. Therefore, the return type of the launch operation in the **ApplicationManager** will support that interface, widened to the CORBA Object type. If a monolithic implementation is being deployed, this comes for free. If the application being deployed is assembly based, however, the **ApplicationManager** has to provide a proxy that supports the expected interface and that delegates invocations on the component interface according to the assembly's specification. Note that this proxy only needs to exist for the top level assembly.

7.9 Miscellaneous

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 operation 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 operation 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.

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.

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.

7.10 Impact on the CCM Specification

This specification is intended to replace the Packaging and Deployment 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 Target Execution Model, 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.

Mapping to XML Schema

8

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

The mapping rules for the XML schemas have been derived from the XML Metadata Interchange (XMI) specification, version 2.0. However, the XML extension mechanism provided by XMI is not needed for this mapping between the PIM constructs and the descriptor schemas. Therefore a mapping to plain XML schemas is performed without inclusion of the XMI schema, but following the XMI strategy.

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, 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 *all* grouping declarator is used for collections of attributes, if no ordering is required. This allows any order of the elements in the resulting document. The minOccurs and maxOccurs schema attributes set to one, protect against omissions of the attribute collection as a whole.

Ordered attributes use the *choice* grouping declarator with maxOccurs set to *unbounded*.

The same method is also used for attributes with multiplicity greater than one in the UML model. But in this case only one attribute is enclosed in the choice block.

“Plural” complexTypes are created for properties and requirements. They use the choice grouping to contain an ordered collection. These collections are then inserted into other types as a single element. The side-effect is the enclosure of the property or requirement collection by a pair of tags reflecting the role name (which is set as the element name).

Associations are mapped using references. [Xlinks should replace these in the next revision of the mapping].

9.1 Overview

The current D&C specification is compliant with the Model Driven Architecture (MDA) defined by the OMG. It is composed of three main levels of models:

- D&C metamodel, which defines the set of metaclasses used in the definition of the D&C specification. The D&C metamodel is defined in relation to the UML 2.0 metamodel.
- D&C Platform Independent Model (PIM), which constitutes the core of the D&C specification. The D&C PIM defines the set of classes and interfaces that are relevant for the implementation of the specification. The D&C PIM is explicitly independent of distributed component middleware (e.g. CORBA or J2EE), information formatting technology (e.g. XML DTD and XML), and programming languages (e.g. C++ and Java).
- Set of D&C Platform Specific Models which constitute implementation of the D&C PIM on concrete platforms. A required CCM PSM constitutes an integral part of this specification. Also, an SCA PSM will be defined for the SWRadio domain.

A PIM-to-PSM mapping is also explicitly defined for each PSM.

While not an explicit part of the current submission, it is also expected 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 UML 2.0, D&C profiles can be defined using the profiling mechanisms provided by UML 2.0.

9.2 D&C Metamodel

The D&C metamodel is aligned with the latest version of the MOF (i.e. MOF 2.0) and UML (i.e. UML 2.0) specifications. The metamodel is composed of three main packages: : BasicConcepts, Component, and Target.

As a naming convention, the name of every leaf level metaclass defined in the D&C metamodel has the suffix “DnC” to explicitly differentiate it from related metaclasses defined in other metamodels.

The BasicConcepts package defines the following concrete metaclasses: DnCProperty, DnCRequirement, DnCDescriptor, and DnCActiveEntity.

The Component package defines the set of metaclasses that are used to model a component-based distributed application. The list of concrete metaclasses currently defined in the Component package includes: DnCComponent, DnCComponentAssembly, DnCConnection, and DnCPort

The Target package defines the set of metaclasses that are used to model a distributed deployment target. The list of concrete metaclasses currently defined includes: DnCArtifact, DnCBridge, DnCDomain, DnCInterconnect, DnCNode, DnCResource, and DnCSharedResource.

The metaclasses defined in the D&C metamodel are used as stereotypes in the D&C PIM.

The metaclasses defined in the Component and Target packages are associated with corresponding “descriptor” classes in the D&C PIM.

9.3 Definition of metaclasses

To be provided.

9.4 Relationship between metaclasses and PIM classes

This section describes the relationship between PIM concepts classes and D&C metaclasses.

Table 2: Component Model Concepts

Concept	Description / proposed Meta-concept / Stereotype	Basic concept	Advanced concept	Meta- concept
several description classes	– meta-concept proposed for all those classes in general concepts table	x		
Package Configuration			x	
Property	– several roles/kinds of properties: +configProperty +selectProperty +execParameter – proposed stereotype: <<DnCProperty>>	x		x
Requirement	– role/kind: +deployRequirement – proposed stereotype: <<DnCRequirement>>	x		x
several reference classes		x		
Implementation Artifact	– proposed stereotype: <<DnCArtifact>>	x		x
Repository Manager	– meta-concept proposed for all manager classes in general concepts table	x		

Table 3: Target Model Concepts

Concept	Description / proposed Meta-concept / Stereotype	Basic concept	Advanced concept	Meta- concept
Domain	– just logical concept	x		
Node	– proposed stereotype: <<dnCNode>>	x		x
Interconnect	– proposed stereotype: <<dnCInterconnect>>	x		x
Bridge	– there may be several (sub)types of bridges such as routers and switches – proposed stereotype: <<dnCBridge>>		x	x
Resource	– proposed stereotype: <<dnCResource>>	x		x
Shared Resource	– special case of resource – so the proposed stereotype for resources <<dnCSharedResource>> also applies for shared resources, i.e. no additional stereotype is needed		x	
Property	– proposed stereotype: <<dnCProperty>>	x		x
Target Manager	– meta-concept proposed for all manager classes in general concepts table	x		

Table 4: Execution Model Concepts

Concept	Description / proposed Meta-concept / Stereotype	Basic concept	Advanced concept	Meta- concept
Deployment Plan	– just an instance of the meta-concept for DnC description proposed in the general concepts table (<<dnCDescriptor>>)	x		
several description classes	– meta-concept proposed for all those classes in general concepts table	x		
Requirement	– role/kind: +deployRequirement – proposed stereotype: <<dnCRequirement>>	x		x
Property	– several roles/kinds of properties: +configProperty; +execParameter – proposed stereotype: <<dnCProperty>>	x		x

Table 4: Execution Model Concepts

Execution Manager	– meta-concept proposed for all manager classes in general concepts table	x		
Application Manager	– meta-concept proposed for all manager classes in general concepts table	x		

Table 5: General Concepts (for all Models)

Concept	Description / proposed Meta-concept / Stereotype	Basic concept	Advanced concept	Meta- concept
Manager	– proposed stereotype: <<dnCActiveEntity>>			x
DnC description	– meta-concept for all three segments/models for description classes (somehow related to XML) – proposed stereotype: <<dnCDescriptor>>			x

Summary: Proposed Stereotypes

So, we propose the following stereotypes:

- <<dnCDescriptor>>
- <<dnCActiveEntity>>
- <<dnCProperty>>
- <<dnCRequirement>>
- <<dnCComponent>>
- <<dnCComponentAssembly>>
- <<dnCConnection>>
- <<dnCPort>>
- <<dnCArtifact>>
- <<dnCBridge>>
- <<dnCDomain>>
- <<dnCInterconnect>>
- <<dnCNode>>
- <<dnCResource>>
- <<dnCSharedResource>>

This section describes the current use case model of the D&C specification. This model is composed of a UML use case diagram and a set of individual use case descriptions.

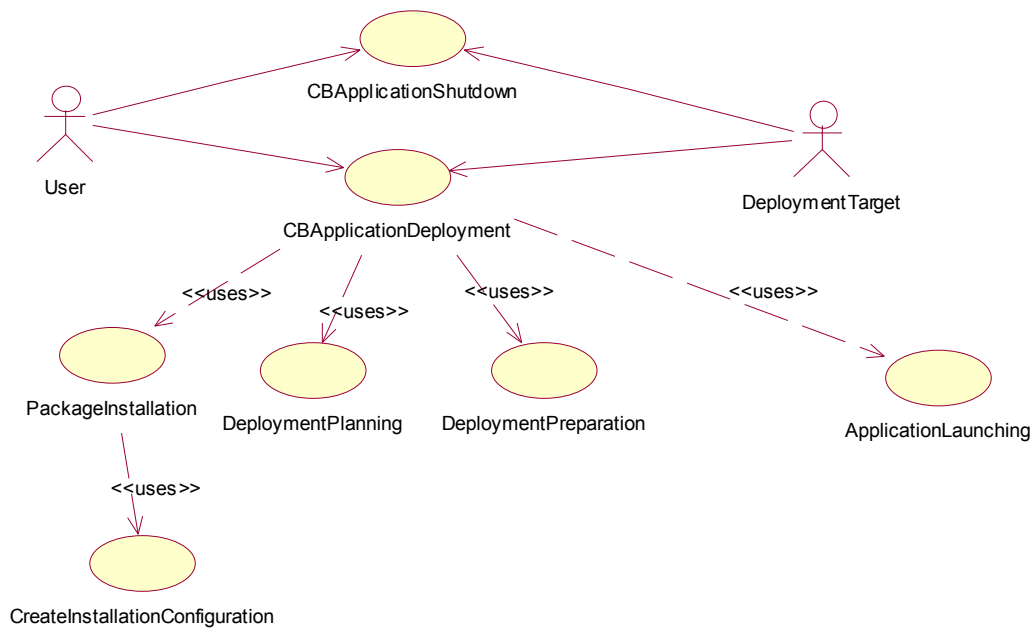


Table 6:

Use Case Identifier: CBApplicationDeployment	Traceability
Description: This use case is a high level description of component-based application deployment.	
External Actors: User, target environment	

Table 6:

Related Use Cases:	
Precondition: Component packages exist and there is a reference available to the user, target environment description is available, target environment is running. Triggering event: User triggers the deployment of a component-based application.	
1. User triggers the deployment of an application. 2. PackageInstallation UC 3. CreatePackageConfiguration UC 3. DeploymentPlanning UC 4. DeploymentPreparation UC 5. ApplicationLaunching UC	
Resulting event: Undefined Postcondition: The component-based application is running.	
Alternatives:	
Nonfunctional requirements:	
Comments:	

Table 7:

Use Case Identifier: PackageInstallation	Traceability
Description: This use case is a description of the steps required to bring component software packages under user control.	
External Actors: User	
Related Use Cases:	
Precondition: Component packages exist and there is a reference available to the user. User has a reference to the repository where s/he wants the packages to be installed. Triggering event: User requests installation.	
1. User requests installation. 2. Packages are copied into the repository. 3. Package is authenticated (optional). 4. Open the package (unzip, error check, decompress, decrypt, virus check, ...). 5. Content specific validation (at this point the package content is in the repository). 6. A default package configuration is created for the installed package. 7. A label is applied to this configuration.	
Resulting event: Undefined Postcondition: The package content and package configuration (virtually) exists in the repository with a label applied.	

Table 7:

Alternatives:	
Nonfunctional requirements:	
Comments: <ul style="list-style-type: none"> - The role of the user is to control the installation of the software component packages. - This use case does not resolve dependencies yet (will come later). - This use case does not customize the installed package. 	

Table 8:

Use Case Identifier: CreatePackageConfiguration	Traceability
Description: This use case describes the step for the creation of a configuration of an installed package (in the repository).	
External Actors: User	
Related Use Cases:	
Precondition: Package content is in the repository. Triggering event: User requests the creation of a new configuration.	
<ol style="list-style-type: none"> 1. User requests the creation of a new configuration. 2. Select the requested package configuration to further customize. 3. Get configuration properties and default values from the package (optional). 4. Get configuration property values from user (overriding default value if #3). 5. Get system deployment property values from user (e.g. those that are not defined by the package, but by the rest of the deployment process). 6. Get a package configuration label from User. 7. Store configuration in repository under supplied label. 	
Resulting event: Postcondition: A new package configuration exists in the repository.	
Alternatives:	
Nonfunctional requirements:	
Comments: Configurations generally refine other configurations, which is why we say that the installation of a package actually creates a default configuration of that package. Another use case could be editing/replacing a package configuration.	

Table 9:

Use Case Identifier: DeploymentPlanning	Traceability
Description: This use case describes the steps to make decisions necessary for the component-based application deployment, which are selections of concrete component implementations and deciding on which nodes these implementations will run.	
External Actors: User (optional)	
Related Use Cases:	
Precondition: Package configuration is available for planning.	
Triggering event: User requests planning.	
<ol style="list-style-type: none"> 1. User request planning for a specific component package configuration label. 2. Obtain static target information. 3. Obtain dynamic target information. 4. Obtain repository search list. 5. Determine a component implementation based on available artifacts, nodes, and resources (also based on a user search policy). 6. Create a deployment plan based on the choice of the step 5. This deployment plan defines the set of artifacts, the nodes on which they must be deployed, the component port connectivity, component configuration properties, configuration property mappings (which actual component gets which property) ... 	
Resulting event: Postcondition: A deployment plan has been created. All deployment requirements must be satisfied.	
Alternatives:	
Nonfunctional requirements:	
Comments: Steps 2, 3, and 4 could be done in any order and multiplicity. This can be partially or completely automated, in which case the user might not be involved. Target information contains static information and optionally dynamic information. Static information is known at the Domain level, while the dynamic information is known at the Node level. Add a comment concerning step 5. Persistence of deployment plan is to be determined. If resources are not reserved in this use case, then there are potential race conditions.	

Table 10:

Use Case Identifier: DeploymentPreparation	Traceability
Description: Deployment preparation is when the target environment is informed of the deployment plan and given the opportunity to perform work and use resources to reduce the effort and time (latency) required to actually launch(start) the application being deployed.	
External Actors: User or deployment management tool.	
Related Use Cases: Deployment planning is a precursor.	
Precondition: A deployment plan and target system must exist.	
Triggering event: User (or tool) requests preparation.	
<ol style="list-style-type: none"> 1. User (or) tool requests preparation 2. User supplies deployment plan, which includes metadata and references to artifacts. 3. User supplies any preparation properties (assuming there are some, like “reusable”). 4. The target system manager (domain manager) accepts, validates and acts on the plan. 5. User receives a “factory” handle that represents a (possibly persistent) preparation. 	
Resulting event:	
Postcondition: the factory exists, and the handle is usable for launching the application	
Alternatives:	
Nonfunctional requirements: The purpose of having this preparation step is to allow the infrastructure to optimize the launch for minimum latency.	
Comments: <p>The preparation step, and the existence of the “factory” uses some resources, but does not necessarily imply that all the resources used during execution are committed or in use.</p> <p>DeploymentFactory.is returned and is used for launching. This allows the location of the factory to be determined by the infrastructure – e.g. it might be best to have the factory in close proximity to the nodes being used for the deployment.</p> <p>One key “preparation property” is whether the factory is intended to be used once or multiple times for launching the application. Knowing that a factory will be used once is important.</p>	

Table 11:

Use Case Identifier: ApplicationLaunching	Traceability
---	--------------

Table 11:

Description: DeploymentLaunch is the step that brings the application to life, bring it to the running state.	
External Actors: User or deployment management tool	
Related Use Cases: Deployment preparation is a precursor	
Precondition: A deployment factory must exist. Triggering event: User (or tool) requests launch	
<ol style="list-style-type: none"> 1. User (or) tool requests launch 2. User supplies deployment factory (object reference returned from preparation) 3. User supplies any launch properties. 4. User receives a “running application” object, which in fact is the reference to an instance of the top level component type, which can navigate to its ports as all components can. 6. User receives a separate “factory creation ID” which can be used to manage the entire instance of deployment. 	
Resulting event: Postcondition: the application is configured and running, and thus the top level port objects are ready to use	
Alternatives:	
Nonfunctional requirements:	
Comments: Any component is “deployable”, thus there is no need to specialize the type of top level components. By using the pattern established in the Fault Tolerant, Data Parallel, Unreliable Multicast, Load Balancing and Reliable Multicast specifications, a separate “factory creation ID” allows lifecycle management without imposing any special type for the “top level” component. If there are many such operations it may be desirable to define a specific management object interface, but that would diverge from the pattern.	

Table 12:

Use Case Identifier: CBAApplicationShutdown	Traceability
Description: DeploymentShutdown is the step that ends the application’s life, bring it out of the running state, releasing all resources used during execution.	
External Actors: User or deployment management tool	
Related Use Cases: DeploymentLaunch is a precursor	
Precondition: A factory creation id must be obtained or retained by the user. Triggering event: User (or tool) requests shutdown.	

Table 12:

<ol style="list-style-type: none"> 1. User (or) tool requests shutdown. 2. User supplies creation/launch ID to DeploymentFactory. 3. Causes all resources associated with execution of application to be released. 4. If factory was single-use, all resources associated with the factory may also be released. 	
Resulting event: Postcondition: the application is not running, its resources are released, object references to the top level component are unusable, factory creation ID is not usable	
Alternatives:	
Nonfunctional requirements:.	
Comments: This does not necessarily imply that resources associated with the factory are released. This brings up the issue of one-time factories vs. reusable factories, and also the issue of how factories are removed. Since the factory is a generic type, the issue of a separate "creation ID" does not necessarily apply. We need to have either a factory method to self-destruct, or a method on the target environment to destroy the factory Releasing resources is implementation dependent. Implementations may have caching strategies that do not visibly release resources even though they are indeed available for re-use.	

Table 13:

Use Case Identifier: DeploymentQuery	Traceability
Description: DeploymentQuery – two steps – get all, then get details. Use same model of data in the plan. Limit to planning information. Two step. Optional stashing. Everything that is running and how it is currently deployed. Snapshot of running system. Factory may be a natural. is the step that ends the application's life, bring it out of the running state, releasing all resources used during execution.	
External Actors: User or deployment management tool	
Related Use Cases:	
Precondition: Target system must be available Triggering event: User (or tool) requests query	
<ol style="list-style-type: none"> 1. User (or) tool requests query of running applications. 2. User receives list of factories. 3. Factories supplies deployment plan information, factory creation ID(s), and component object. 	
Resulting event: Postcondition: user or tool has information	
Alternatives:	

Table 13:

Nonfunctional requirements:.	
Comments:	

11.1 Summary of optional versus mandatory interfaces

All interfaces are mandatory.

11.2 Proposed compliance points

Each PSM is its own compliance point.

11.3 Changes or extensions required to adopted OMG specifications

As intended, the CCM PSM replaces the deployment and configuration chapters of CCM 3.0.

11.4 Complete IDL definitions

Note that IDL definitions for the CCM PSM are generated based on rules.

References

12