Automated Deployment of Component-based Applications into Heterogeneous Distributed Systems

Frank Pilhofer, James E. Kulp
Mercury Computer Systems, Inc.
Chelmsford, MA, U.S.A.

Andreas Hoffmann
Fraunhofer FOKUS
Berlin, Germany

June 16th, 2004

Abstract

Emerging component-based distributed architectures promise better re-use of software components, by manufacturing applications by assembling existing parts.

This paper presents an overview of the “Deployment and Configuration of Component-based Distributed Applications” specification that was adopted by the Object Management Group last year, defining a model for creating and packaging hierarchical, component-based applications, and interfaces for deploying applications into heterogeneous distributed target systems. Design choices are explained and elaborated.

The specification extends the application and deployment models of existing component-based distributed architectures like the CORBA Component Model (CCM) and the Software Communications Architecture (SCA). Based on the annotation of software with meta-data in XML format, it allows the re-use of individual components as well as of assembly compositions, and addresses the challenge of modeling implementation requirements and their matching against hardware resources, to allow a fully automated deployment process to assign components to computing nodes.

1 Introduction

Component-based distributed architectures gained popularity in the late 1990’s with the success of Enterprise Java Beans (EJB) and Java 2 Enterprise Edition (J2EE), followed by the introduction of the programming language-independent CORBA Component Model (CCM). These architectures targeted multi-tier enterprise systems, where components were used in a middle tier, representing database objects and being used by a Web client.

Technologies to distribute applications across a heterogeneous network as a means for increased performance date back to software such as the Parallel Virtual Machine (PVM). Yet such distributed applications were, in a sense, still monolithic, as individual “components” were usually not designed for reuse. Assignment of components to hosts and data distribution was manual.

Going beyond EJB, the CCM introduced the notion of component “ports.” Multiple components could then be interconnected via these ports into an “assembly.” An assembly could then be deployed as a standalone application. This was achieved using meta-data-XML files with information about components and assemblies, allowing generic tools to create, package and deploy such component-based applications. However, CCM’s capabilities for automated distribution of components across nodes were insufficient.

Recently, the Lightweight CORBA Component Model (LwCCM) was published as a profile of the “full” CCM. By removing enterprise-level features such as transactions, persistence and introspection, it is more oriented...
towards the purpose of distributed applications running within a networked embedded system.

Even though component technology is not limited to embedded systems, it is recently being embraced by this domain, as embedded systems become more heterogeneous. As diverse hardware, including DSPs and FPGAs, is being integrated with general-purpose processors, middleware such as CORBA is required to ensure interoperability. Applications need to take the best possible advantage of rapidly changing hardware without manual reconfiguration. Customers demand commercial off-the-shelf rather than proprietary solutions for the integration of software.

As evidence of this, the Joint Tactical Radio Systems (JTRS) program mandates compliance of future US military software-defined radio systems with the Software Communications Architecture (SCA) [5], a CORBA-based, component-centric middleware, with the intention that radio software be portable to a wide range of hardware, and interoperable with other software. The software-defined radio market, with its high computational requirements combined with space and power constraints, is now a driving force behind component-based middleware.

The “Deployment and Configuration of Component-based Distributed Applications” (D+C) specification [2] was conceived in part to reconcile both efforts, expanding on ideas from both CCM’s packaging and the SCA’s deployment model, resulting in a specification that addresses the modeling, packaging and deployment of component-based distributed applications as a superset of either architecture.

2 Overview

The user point of view of the D+C specification is concerned with the various actors that are involved in the development and deployment of component-based software. While it is these actors that eventually “drive” deployment, it is the interfaces used by the actors, and the data that is exchanged, that need to be defined by the specification. This provides a second, “model” point of view, with different meta-data models to describe component-based software, hardware capabilities, and “deployment plans” detailing a specific execution scenario. The model also includes the interfaces that then perform deployment based on the meta-data.

A third “vendor” point of view defines individual compliance points for building blocks, recognizing that pieces of infrastructure may be supplied by separate vendors and thus need to interoperate. This includes development tools, different pieces of deployment infrastructure, and the hardware-specific operating environment.

2.1 Scope

An important step in the development of a specification is the limitation of its scope. At the beginning of the process, the most important requirements identified were:

- A platform-independent and vendor-independent description of component-based applications, using meta-data.
- Support for alternative implementations of the same component, so that an appropriate implementation can be chosen at deployment time to match available hardware.
- Expression of the application’s requirements and the target system’s resources, to allow a generic, automated matching process between them.
- Separation of concerns between different actors—i.e., developers, users, and vendors—with well-defined interfaces between them.

The D+C specification limited its scope to the startup—with an initial configuration—and termination of applications, and does not address any runtime behavior. For example, the issue of starting additional components within a running application was declared out of scope, but may be addressed by future add-on specifications.

2.2 Model-driven Architecture

The D+C specification uses model-driven architecture (MDA) [4] concepts to define its meta-data models. A platform independent model (PIM) is refined into platform specific models (PSM). The PIM is independent of the component architecture, assuming only a generic notion of components, with ports and attributes, as discussed
Both PIM and PSM use the MOF subset of UML to describe meta-data elements and their associations. While UML diagrams are ideal to document a model, more “concrete” representations are needed to transport meta-data. For this purpose, the model is eventually mapped, using the UML Profile for CORBA [7] and the XML Metadata Interchange [9] specifications, to IDL on one hand, and to an XML Schema on the other hand. As both specifications require differently-augmented, incompatible input models, a small intermediate step transforms the original model into a suitable source for either.

The full set of model transformations and mappings is shown in figure 1. All of them are implemented in a custom plugin for IBM Rational Rose, thereby keeping model and model-generated files in sync.

At runtime, meta-data can be represented by IDL defined data structures, and by XML files for “offline” use. Since both IDL and the XML Schema are automatically generated from a common model, transformations between the two representations, i.e., XML parsers and serializers, can be automatically generated.

2.3 Standards Use

One design decision was to use existing standards where domain-specific solutions were not necessary. The UML Profile for CORBA and XML Metadata Interchange were already mentioned.

The Extensible Markup Language (XML) [10] is used for the off-line representation of meta-data as a good compromise between human and machine readability. Easily generated and parsed by software, XML files can also be read and manipulated in a text editor if necessary.

Within the meta-data model, Universal Resource Identifiers (URIs) [8] are used as both locators (i.e., URLs) and names (i.e., URNs). Locators are used within XML files, as cross-references as well as pointers to both intra- and extra-package artifacts and other packages. Names are used to uniquely identify packages and artifacts; they can be used to request specific package versions, or for artifact caching.

The de-facto ZIP standard [11] was chosen as the file format for component packages, i.e., to distribute component-based software between developers and users of the software. ZIP files allow for single-file software distribution in a machine-independent format.

As part of the deployment infrastructure, the Hypertext Transfer Protocol (HTTP) [3] is used as the default means of distributing files to nodes, for execution. Chosen as a flexible, ubiquitous, well accepted standard over a self-defined file transport mechanism, many implementations are readily available. Simple embedded HTTP clients and servers are easy to implement; more complex versions can support proxies and introspection—e.g., timestamps to not download an artifact if it was not modified.

3 Component-based Software

While the D+C specification is about software “deployment,” a model of the software being deployed is required. Together with the actual deployment process, this “component model” is at the heart of the specification, as it defines the scope of “deployable” software.
3.1 Component Interface

The basic entity in component-based software is a component.

“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 […] wiring together their required and provided interfaces.”

Figure 2 shows an illustration of a component that has one provided interface port, two used interface ports, and two attributes. The component is shown as a black box, entirely defined by its set of ports and attributes, to imply that the component’s implementation is replaceable, as long as each implementation conforms to the same interface.

3.2 Component Implementation

Once a component interface is specified, one or more component implementations of that interface can be created. Based on the same set of syntactic and semantic requirements, they are functionally identical. However, they may differ in terms of quality of service. Some implementations might be limited to certain hardware configurations, but provide “better” service, such as less latency. The D+C specification defines a grammar to express these orthogonal properties: requirements on one hand, to be matched by available hardware and software, and implementation capabilities, to support the selection of a particular implementation by a user of the software.

A component can be implemented in two discrete ways, monolithic or assembly-based.

3.2.1 Monolithic Implementation

A monolithic implementation is an executable piece of software consisting of a set of opaque artifacts (i.e., files), as illustrated in figure 3.

In the D+C specification, monolithic implementations may express requirements that need to be matched against available hardware and software resources. This may include dependencies on software (e.g., operating systems), hardware (e.g., CPU type), or devices accessed by the implementation, such as a sound card. The matching process is performed at deployment time, when the actual set of available (and unused) resources is known.

The term executable is not elaborated; it is assumed that the infrastructure on the target computer knows how to create a running component instance from the set of files (see section 5.3 below). Usually, the implementation will include a requirement that implies a service contract with the computer’s infrastructure.

Therefore, it does not matter to the deployment infrastructure whether a file is a Windows executable, a shared library, or a chunk of FPGA firmware, all it does is deliver the artifacts to the target computer.

3.2.2 Assembly-based Implementation

An assembly is a set of interconnected subcomponents as shown as a “white-box” view in figure 4. An assembly-based implementation implements a component interface by “mapping” the interface’s ports and properties to compatible subcomponent features. The effect is a wiring di-
Figure 4: An assembly is a set of interconnected subcomponents like in circuit design, where a chip’s external pins are connected to submodule pins. Thus, an assembly is fully modular and reusable. To users, i.e., to other components interacting with the assembly via its ports and properties, its internal composition is entirely transparent.

Assemblies do not depend on particular subcomponent implementations, as long as they satisfy any QoS requirements that an assembly may place on subcomponents. An assembly recursively points to component packages (see section 3.4), by name or by reference, to provide alternative implementations for subcomponents. Some subcomponents may have monolithic implementations, while others are assemblies. The tree defined by an “application” component at its root, assemblies as inner nodes with subcomponents as child nodes, and monolithic implementations as leaf nodes, is not necessarily balanced.

Connections between ports may be annotated with requirements. The intention is to express, and later to match against available hardware, connection-related quality of service like bandwidth or latency.

Assemblies are hardware and software independent, and thus do not express target requirements. Instead, an assembly’s requirements are implied by available implementations for its subcomponents. Of course, at deployment time, subcomponents can be spread across multiple nodes to take advantage of the “best” implementation for each subcomponent—as long as the transport between nodes can satisfy the requirements placed on the connections.

### 3.3 Component Model

The component model defines a set of meta-data elements that are used to describe software according to the concepts presented in the previous sections. This meta-data can then be used by generic development and deployment tools. Figure 5 shows a UML diagram of the meta-model elements and their associations.

The Component Interface Description element describes a component interface as defined in section 3.1. The Component Implementation Description element describes a component implementation to be either monolithic or assembly-based, as defined in section 3.2. The Monolithic Implementation Description element describes a monolithic implementation as a set of implementation artifacts, as defined in section 3.2.1. Finally, the Component Assembly Description element describes an assembly-based implementation as a set of interconnected subcomponents as defined in section 3.2.2.

On top of these elements, the D+C specification adds the Component Package Description element. It describes a set of one or more alternative implementations of the same component interface. Implementations are assumed to be semantically equivalent and thus replaceable. At deployment time, one of the alternative implementations can be chosen based on available resources and policies.

Note that the assembly meta-data does not point to specific implementations for its subcomponents, but again to a package description, thus being independent of available subcomponent implementations. Assemblies can take immediate advantage of new implementations that are added.
to a subcomponent’s package.

Finally, a software package contains a single top-level Package Configuration element, which is used to configure the software regardless of which of the alternative implementations will be chosen at deployment time. A Package Configuration can itself be recursive, refining an existing configuration, but ultimately references a Component Package Description “base package.”

### 3.4 Component Package

A component package is a redistributable piece of software. The ZIP file format is used as a container for a set of implementation artifacts and descriptive meta-data in XML file format. As a compressed single file, component packages can be easily distributed by file transport, download or e-mail.

A package may be an aggregation of implementations as well as contents from an assembly’s subcomponent packages. It might contain multiple Package Configuration descriptors. As a means to identify the top-level package, the ZIP file is defined to contain a single magic “package.tpd” file pointing to the Package Configuration of the outermost component.

Component packages are optionally, but not necessarily self-contained. For example, assemblies may reference external packages, to allow for the independent evolution of subcomponent implementations.

### 3.5 Development Actors

Various development actors collaborate in creating a component package. Their motivation is to create component-based applications that are able to adapt to a variety of heterogeneous distributed systems. Figure 6 shows an overview of the four actors that are involved in software development.

First, a Specifier actor defines a component interface. This interface is implemented either monolithically, by the Developer actor, or assembly-based, by the Assembler actor. While an assembly is a meta-data construct only, the developer also produces a number of executable implementation artifacts. The Assembler uses existing component packages, by composition or by reference, to provide subcomponent implementations.

#### Figure 6: Development Actors

The Packager actor then creates a component package containing multiple monolithic or assembly-based implementations.

### 4 Hardware and Software Resources

The component model allows developers to annotate component-based software with requirements that are to be matched against available hardware and software. Monolithic implementations can express requirements that need to be matched by the node that they will execute on (section 3.2.1), and assemblies can express requirements for its connections (section 3.2.2), to be matched by the intra- or inter-node transport mechanism. Consequently, the D+C specification defines a “target model” to express hardware and software resources.

#### 4.1 Target Model

The basic entity in the target model is the node, which represents a target for the execution of monolithic implemen-
tations. In symmetry with the opaque nature of executable artifacts, no assumptions are made about the actual type of node. The concept encompasses workstations as well as embedded co-processors or FPGAs, i.e., any device that is capable of executing components.

Multiple nodes may form a network. Direct connections between nodes are called interconnects (e.g., an Ethernet cable), connections between interconnects are called bridges (e.g., an ethernet switch).

In embedded systems, it is common to have hardware devices, such as I/O equipment (e.g., a modem), that can be accessed from multiple nodes. Such devices are recognized as shared resources.

These four kinds of entities, nodes, interconnects, bridges and shared resources, make up the target environment, which is called the domain.

Like the component model, the target model (see figure 7) defines a set of meta-data elements to describe the composition of a domain. As explained in the previous section, the domain is composed of a set of nodes, interconnects, bridges, and shared resources. Interconnects provide direct, and bridges provide indirect connectivity between nodes, i.e., connections between components that run on indirectly connected nodes need to be routed across multiple interconnects and bridges, and thus their requirements must be matched against all their capabilities.

Nodes, interconnects and bridges each describe a set of resources. Shared resources are attached to one or more nodes, and are thus available to components executing on either.

4.2 Resource Model

Figure 8 shows the meta-data representation for resources, which is used by the target model. Resources are identified by one or more “resource type” URNs—a single resource may support more than one resource type. Each specific resource type contains a well-defined set of “satisfier properties,” i.e., a property that can satisfy a particular required property. For example, an “operating system” resource might define “name” and “version number” satisfier properties.

Satisfier properties support various types of accounting and predicates, enumerated in figure 8 as the Satisfier-PropertyKind type, to compare a required value against the resource’s capacity value. For example, a resource of Attribute kind is not consumable (e.g., the name of an operating system), whereas a resource of Capacity kind is used up as components are deployed (e.g., the amount of available memory).

By defining the grammar for expressing generic requirements, and an algorithm for consuming resources, the D+C specification allows generic software to match requirements against resources.

Additional specifications are assumed to define specific “vocabularies” of resource types at the PSM level, with well-defined sets of property names and kinds.
4.3 Target Manager

The target manager is a runtime service, with a well-defined interface, that centrally maintains a domain’s meta-data. It provides operations for retrieving the meta-data, with resources either reflecting their total (initial) or remaining available capacity. Two more operations support the commitment of resources, when software is being executed, and the release of resources when the application terminates. A fifth operation allows for meta-data updates at runtime, if necessary, e.g., when a new node is added to the system.

The target manager is a stand-alone entity, independent of the actual nodes. It maintains their meta-data, but does not necessarily interact with them. This allows off-line scenarios, in which clients can match application requirements against a known state of the domain, using the same interface that will be used in an on-line situation.

5 Deployment

The deployment process takes a component package, and arranges for the execution of an application within a domain. Deployment can be separated into three phases, installation, planning and execution. The corresponding actors are shown in figure 9.

5.1 Installation

Within a user’s domain, the Repository Administrator installs deployable software in one or more “repositories.” The “repository manager” is a service that allows installation, retrieval and deletion of packages, as delivered from the Packager actor.

Repositories act as a staging area under the user’s control, and they provide central access to package meta-data and artifacts. Within a repository, application properties can be configured with custom preferences, e.g., a uniform background color for graphical applications.

When a package is installed in a repository, the repository manager parses the package’s meta-data and artifacts. Within a repository, application properties can be configured with custom preferences, e.g., a uniform background color for graphical applications.

By taking in XML meta-data and ZIP-format packages, and giving out IDL-defined data structures and HTTP URLs, the repository manager is the only entity of the deployment infrastructure that needs to understand the XML and ZIP formats. Clients, such as the individual nodes with potentially tighter footprint constraints, only need an ORB and a comparatively small HTTP client (or a different, agreed-on file transport protocol).

5.2 Planning

The planning process prepares an application for execution within a particular domain. The Planner retrieves meta-data for the package that is to be deployed from the repository, using the repository manager interface. Referenced packages, e.g., for implementations of subcom-
ponents in an assembly, are resolved using a search path of repositories. The planner then matches implementation requirements against a specific domain’s resources, which are retrieved from a target manager, as described in section 4.2.

A Planner has to find a valid deployment, i.e., an assignment of monolithic implementations to nodes, so that on each node, available resources match or exceed the sum of requirements. Because packages may contain multiple alternative implementations, a Planner may have to recursively consider multiple decompositions of an application into monolithic components.

The D+C specification describes a non-normative planning algorithm, but does not mandate a particular implementation, allowing for planning to be a quality of implementation issue. Considering the number of permutations, an exhaustive traversal of the search space may not be practical. A simple planner might fail if it does not find a valid deployment right away, while another might keep on searching. While automated planning is possible, a planner implementation could also be interactive, allowing users to make component assignment decisions manually.

The specification does not define a metric to compare deployments and thus does not have a notion of an “optimal” deployment. However, planner implementations may have preferences, e.g., to spread implementations across as many or as few nodes as possible.

The planner’s output is a deployment plan, which is a self-contained data structure with all necessary information to execute the application, based on decisions made during planning. The plan represents a “flat” assembly that only contains components with a monolithic implementation, and the connections between them. For each component, it names the node on which it is to be executed, and the artifacts that are part of its implementation.

Planning can be done “on-line,” based on currently available resources, or “off-line,” based on a future, known state of the domain; in the trivial case, a domain with no running applications. On-line planning is necessary if the domain’s state is dynamic, and resource allocation is thus unknown ahead of time. On the other hand, a common use case in embedded systems is a small number of applications that are executed, exclusively or not, in a predictable fashion. This allows to perform the potentially expensive planning process off-line, and then to store the set of resulting deployment plans for later use.

Note that planning, once the application’s meta-data is retrieved from the repository manager, and the available resource information is retrieved from the target manager, is an entirely “local” process, as no further remote procedure calls are necessary. In other component middleware like the SCA, numerous and potentially repetitive (in case of backtracking) roundtrips to each node result in high latency. Race conditions, when the deployment of other applications uses resources while a planner plans to also commit them, exist in either scenario and may have to be resolved, if necessary, by a global transaction mechanism.

5.3 Execution

Finally, the application is executed, based on a deployment plan. The plan can be seen as a precise “script” that directs which components to execute where, so there are no more planning decisions to be made. Execution involves

- committing resources (using the target manager),
- downloading artifacts onto nodes (from the repository manager),
- loading artifacts into memory,
- instantiating components,
- configuring components,
- interconnecting components, and
- starting all components.

The execution manager is a singleton run-time service for the execution of deployment plans within a domain. It delegates operations to node manager services on each node. For this, the plan is decomposed into a set of partial component assemblies, so that each executes entirely within a node. This “virtual” component’s external ports are then implied by connections across nodes. The node manager interface again supports execution based on a deployment plan, as deployment of these intra-node assemblies can be described using the same data structure—with the constraint that the target node is the same for each component.
While the execution manager is a generic piece of infrastructure, node managers are specific with respect to their node type, completing the circle that begins with the developer actor (see section 3.5). Tools to develop monolithic component implementations (e.g., compiler and ORB), and node managers capable of executing self-developed artifacts, will usually be provided by the same vendor. A well-defined interface between the execution manager and the node managers ensures interoperability across heterogeneous nodes.

A node manager service is not necessarily collocated with the managed node itself. It could legitimately run on a different node, as long as it had the capability to launch component implementations on the node in question. This allows highly specialized nodes such as DSPs or FPGAs to be part of a domain, even if they do not host an operating system or are not capable of running concurrent processes.

Before component instances are launched, a separate “preparation” phase takes place, which might encompass any of the first four steps in the above list. Again, this is implementation specific, and should be user tunable. A likely implementation will at least download artifacts onto nodes, but node managers could also go further, so that the application can then be started with the smallest latency possible.

Preparation, and later the interconnection of components, is designed for maximum concurrency. In a first step, the execution manager splits up the deployment plan and sends its pieces to each node manager. Secondly, the execution manager retrieves references of “provided” ports exported by each component. In a final third step it then broadcasts connections to “used” ports. Each step can be performed concurrently, with the execution manager sending information to all node managers at once, and then collecting their replies. Requiring only three sequential roundtrips dramatically reduces startup latency.

6 Summary

The D+C specification enables component-based distributed applications. It supports the automated deployment of such applications into heterogeneous distributed target systems. Components, whether assembly-based or not, are independent, reusable work products. With these features, D+C is a leap forward from deployment in CCM, EJB, or SCA.

When the CORBA component model is extended, e.g., with stream ports, or DDS ports based on the OMG’s Data Distribution service, it will be easy to support such additional features within the D+C framework.

The D+C specification may be applied to a wide range of component models, beyond CCM. Following the MDA approach, the core of the specification is a platform-independent model that defines the deployment machinery, including interfaces and exchange formats for metadata. This supports developers in following the MDA principles while developing and deploying their heterogeneous distributed applications, allowing easy porting of applications once further mappings of the D+C PIM to other platforms are available.

In future, the target model could be refined to better support large-scale domains, so that individual “node descriptors” could be referenced, rather than replicated, for identical, homogeneous sets of nodes. Ideally, such descriptors would then be provided by hardware vendors.

At the moment, the D+C specification’s scope is limited to development and deployment time. Extending its concepts into an application’s runtime would be interesting research, e.g., to support “mutating” applications where individual components are started and stopped as necessary, component migration, or fault tolerance.

References


June 2003.


